An Introduction to Unsupervised Learning

Alex Young and Cenhao Zhu

Version 0

Contents

Chapter 1

Introduction

Unsupervised machine learning (UL) is an overarching term for methods designed to understand the patterns and relationships within a set of **unlabeled data**. UL is often discussed in contrast to (semi-)supervised learning. In the latter setting(s), one is primarly concerned with prediction and classification, and the machine learning algorithms therein focus on learning the relationship between a (often high dimensional) feature vector $\vec{x}$ and an observable outcome y by training on a labeled dataset $\{(\vec{x}_i, y_i)\}_{i=1}^{N}$. As a concrete example, consider the MNIST dataset which contains a compendium of labeled digitized grayscale images of handwritten digits [**?**]. The images themselves are the features, $(\vec{x})$, and the identity of the digit (0 to 9) gives the label, y . A supervised learning algorithm trained on the MNIST dataset would be able to classify a handwritten digit in a new grayscale image.

Prediction and classification are clearly defined goals which naturally translate to many settings. As such, supervised ML has found numerous applications across diverse fields of research from healthcare and medicine to astronomy and chemistry. Given the clearly translatable goals of supervised learning, most texts on Machine Learning tend to emphasize this setting with much smaller discussion on the un- or semi-supervised setting. For example, both *The Elements of Statistical Learning* by Hastie et al [**?**] and *Modern Multivariate Statistical Techniques* by Izenman [**?**] are wonderful texts – which were central to the early development of this book – but lean towards supervised problems.

Unlike the supervised setting, however, UL algorithms are applied to datasets without (or ignoring) labels. In contrast to the MNIST example above, you can think of have of a case where one has access to a large collection of $\vec{x}_i$, such as images, without any labels indicating the content(s) of the image. Other examples include,

- a corpus of emails without any indication of which, if any, are spam
- genomic data for each individual in a large population of cancer patients

- collections of consumer data or ratings

Without labels, it may be difficult (particularly to students first seeing this branch of machine learning) to grasp the usefulness of UL including its applicability and what one is *learning* in practice. In this book, we hope to address this difficulty and provide readers with a clear understanding of UL by covering motivating ideas, fundamental techniques, and clear and compelling applications. For now, we'll discuss the high-altitude view of unsupervised learning.

We'll focus on those cases where we have a collection of independent observations of (preprocessed) features stored in vectors $\vec{x}_1, \ldots, \vec{x}_N \in \mathbb{R}^d$. This setting will be formalized in 2. Broadly, UL learns patterns and similarities between the vectors which could allow us to

1) find subsets of the data which more similar to each other (clustering) or
2) find simpler representations of the data which preserves important relationships (dimension reduction)
3) identify common relationships between variables in the data (association rules)

Each of three cases provide a simplified lens through which we can view our data, and in doing so, can open up a number of interesting possibilities.

Clustering different genomic data in cancer patients could provide information to medical practitioners on which cancers exhibit common genetics signatures. Applying dimension reduction to spam emails could allow one to identify odd emails which might be spam (anomaly detection). Learning the common relationships between variables in a consumer data set opens the possibility of matching consumers which items they might enjoy (recommendation systems).

The examples above are by no means exhaustive, but they do raise a few critical points.

1) **Where** Each application above is one step in a larger data science problem. Unsupervised learning is rarely detached from a broader data science pipeline. This is a stark difference from classification and prediction which are often viewed as isolated statistical problems (though careful practitioners recognize that data collection and cleaning and the communication of results are often of equal or greater importance than analysis). Examples are provided throughout the text demonstrating where UL can be useful.

2) **What** One data set could be approached from one or more different perspectives. For example, one could apply dimension reduction to the cancer data to *visualize* the potentially complex data in a manner that preserves important relationships. Combining many approaches together makes unsupervised learning a powerful tool to exploratory data analysis and featurization, particularly when combined with expert level content knowledge. Choosing a UL method is linked with what we hope to learn about our data.

3) **How** If we want an algorithm that clusters similar vectors or provides a visualization that keeps close points together, then we should be mindful of the meaning of similarity or proximity. UL algorithms, sometimes implicitly, prioritize different relationships. We explore how these algorithms work from a geometric perspective which is a helpful intellectual scaffolding.

In the remainder of this text, we focus primarily on dimension reduction (4 and 6) and clustering (7).

1.1 Prerequisites

This text is targeted at upper level undergraduates with a well rounded background in the following courses and topics

1) Probability: random variables, expectation, variance, and covariance
2) Linear algebra: matrix-vector multiplication, linear spaces, eigendecompositions
3) Multivariable calculus: gradients and basic optimization

A brief review of the most important ideas is covered in Chapter 2. Additional tools and techniques needed for specific algorithms are covered at a cursory level as needed. References to more thorough discussions are provided throughout for the interested reader.

Chapter 2

Mathematical Background and Notation

This text assumes a familiarity with probability theory, multivariable calculus, and linear algebra consistent with what one would see in an undergraduate course. The most important concepts are

- Probability:
  - Major concepts: mean, covariance, correlation, (in)dependence
  - Minor concepts: likelihood functions, conditional expectation, parametric distributions (e.g. Gaussian)
- Linear Algebra
  - Major concepts: interpretations of matrix-vector and matrix-matrix multiplication, spans, bases, matrix decompositions
  - Minor concepts: orthonormal transformations, vector and matrix norms, Hadamard operations
- Multivariable calculus:
  - Major concepts: gradients (computation and interpretation), optimization with and without constraints
  - Minor concepts: multivariate Taylor expansions, determing properties of local optima via Hessian matrix

In this chapter, we will identify common notation and review the most important prerequisite material.

2.1 Important notation

Throughout this text, we will be working with vectors and matrices quite often so we begin with a bit of notation and a few important conventions we will adopt hereafter. We'll use notation $\vec{x} \in \mathbb{R}^d$ to denote a d-dimensional vector.

Importantly, we adopt the convention that vectors are columns vectors by default so that

$$\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}$$

where $x_1, \ldots, x_d$ are the entries or coordinates of vector $\vec{x}$. Row vectors are then the transpose of column vectors so that $\vec{x}^T = (x_1, \ldots, x_d)$. When needed we'll let $\vec{0}$ denote a vector of all zeros and $\vec{1}$ a vector of all ones with the dimensionality defined implicitly, e.g. if $\vec{x} \in \mathbb{R}^d$ then in the expression $\vec{x} + \vec{1}$, you may interpret $\vec{1} \in \mathbb{R}^d$ so the summation is well defined.

Matrices will be denoted in **bold** so that $\mathbf{A} \in \mathbb{R}^{m \times n}$ denotes an $m \times n$ matrix with real entries. Subscripts are read as *row,column* so that $\mathbf{A}_{ij}$ is the entry of $\mathbf{A}$ in the ith row and jth column. A superscript T denotes the transpose of a matrix. For a square matrix $\mathbf{B} \in \mathbb{R}^{n \times n}$, we use notation $Tr(\mathbf{B})$ to denote the trace of $\mathbf{B}$ and $det(\mathbf{B}) = |\mathbf{B}|$ to denotes its determinant.

Using this above notation, we may also define the inner product and outer product of two vectors. For vectors $\vec{x}$ and $\vec{y}$, the inner product or dot product of $\vec{x}$ and $\vec{y}$ is the scalar $\vec{x}^T\vec{y} = \sum_{i=1}^{d} x_i y_i$. Alternatively, we may also consider the outer product $\vec{x}\vec{y}^T$ which is a matrix such that $(\vec{x}\vec{y}^T)_{ij} = x_i y_j$. For the inner product to be well defined $\vec{x}$ and $\vec{y}$ must have the same dimension. This is not the case for the outer product. If $\vec{x} \in \mathbb{R}^m$ and $\vec{y} \in \mathbb{R}^n$ then $\vec{x}\vec{y}^T \in \mathbb{R}^{m \times n}$. If we view a d-dimensional vector as a $d \times 1$ matrix, then both of these algebraic computations are completely consistent with standard matrix multiplication which we will revisit near the end of this chapter.

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a function of d-dimensional vector $\vec{x}$. Then we define the gradient of f with respect to $\vec{x}$, denoted ∇f, to be the d-dimensional vector of partial deriviates of f with respect to the coordinates of $\vec{x}$ so that

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_d} \end{bmatrix}.$$

The Hessian matrix of f with respect to $\vec{x}$, denoted as $\mathcal{H}f$, is the $d \times d$ matrix of second order partial derivatives of f with respect to the coordinates of $\vec{x}$ so that $(\mathcal{H}f)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$. If we are considering a function of multiple vector valued variables, e.g. $f(\vec{x}, \vec{y}, \vec{z})$ then we use $\nabla_{\vec{x}}$ to denote the gradient of f w.r.t. the vector $\vec{x}$ only.

2.2 Random vectors in $\mathbb{R}^d$

Throughout this text, we will consider independent, identically distributed (**iid**) samples of a d-dimensional random vector $\vec{x} = (x_1, \ldots, x_d)^T$. Each coordinate

x_i is a random variable and we may view the distribution of the random vector $\vec{x}$ as the joint distribution of all of its coordinates. The cumulative distribution function of $\vec{x}$ is then

$$F(\vec{x} \leq \vec{x}^o) = P(\vec{x} \leq \vec{x}^o) = P(x_1 \leq x_1^o, \ldots, x_d \leq x_d^o).$$

We'll largely consider continuous entries so we can rewrite the above in terms of the joint density, $f : \mathbb{R}^d \to [0, \infty)$, of $\vec{x}$ such that

$$F(\vec{x} \leq \vec{x}^o) = \int_{-\infty}^{x_1^o} \cdots \int_{-\infty}^{x_d^o} f(x_1, \ldots, x_d) dx_d \ldots dx_1.$$

To simplify notation, we'll often write the above as

$$\int_{-\infty}^{x_1^o} \cdots \int_{-\infty}^{x_d^o} f(x_1, \ldots, x_d) dx_d \ldots dx_1 = \int_{-\infty}^{\vec{x}^0} f(\vec{x}) d\vec{x}.$$

In the case that $\vec{x}$ may only take one of a countable set of outcomes, one can replace the integral above with a corresponding summation.

Generally speaking we will be considering data drawn from an unknown distribution. However, considering known cases which we can analyze and sample from is often helpful to study how different algorithms perform. With this idea in mind, let's define a few different distributions which we will revisit throughout this chapter as examples. In each case, we will also provide scatterplots of independent samples from these distributions so that you can visualize the distributions more directly.

Definition 2.1 (Multivariate Gaussian Distribution)**.** The multivariate Gaussian distribution in $\mathbb{R}^d$ with mean $\vec{\mu} \in \mathbb{R}^d$ and symmetric positive definite covariance matrix $\mathbf{\Sigma} \in \mathbb{R}^{d \times d}$ is the random d-dimensional vector $\vec{x}$ with probability density function

$$f(\vec{x}) = \frac{1}{(2\pi)^{d/2} det(\mathbf{\Sigma})^{1/2}} \exp\left(-\frac{1}{2}(\vec{x} - \vec{\mu})^T \mathbf{\Sigma}^{-1} (\vec{x} - \vec{\mu})\right).$$

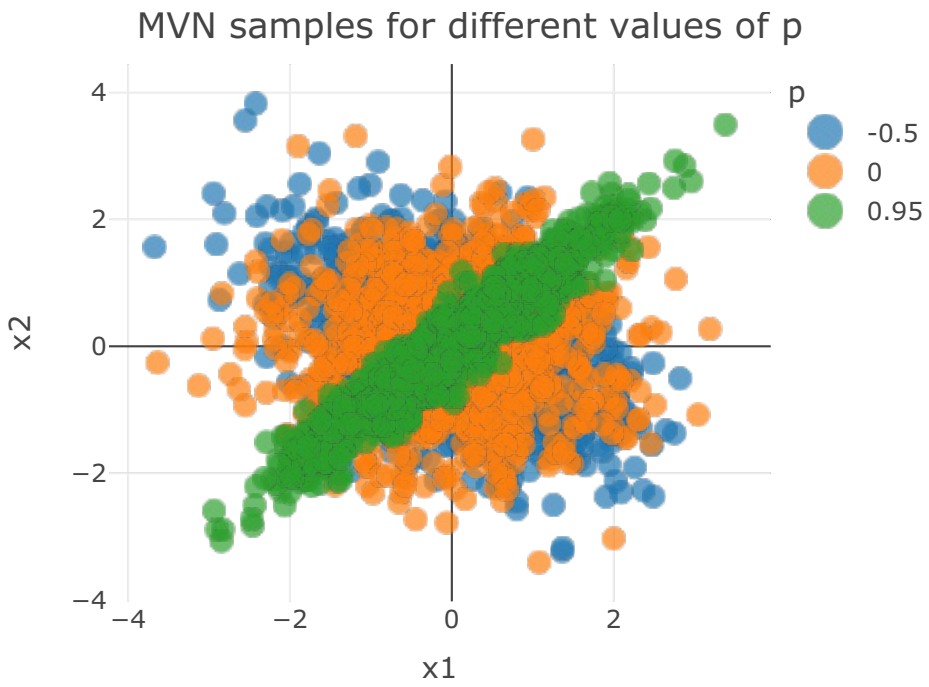
We use shorthand $\vec{x} \sim \mathcal{N}(\vec{\mu}, \mathbf{\Sigma})$ to indicate $\vec{x}$ follows this distribution.

The Multivariate Gaussian distribution is also often called the Multivariate Normal (MVN) distribution.

For example of the MVN, first consider the two-dimensional case with $\vec{\mu} = \vec{0}$ and

$$\mathbf{\Sigma} = \begin{bmatrix} 1 & p \\ p & 1 \end{bmatrix}.$$

Below, we show scatterplots of 1000 independent samples from this distribution for three different values of p. We will also refer to a collection of points in $\mathbb{R}^d$ as a point cloud.



For an examples in $\mathbb{R}^3$ we again consider case where $\vec{\mu} = 0$ and let

$$\mathbf{\Sigma} = \begin{bmatrix} 1 & p & p^2 \\ p & 1 & p \\ p^2 & p & 1 \end{bmatrix}.$$

In the preceding examples, different choices of p, hence different covariance matrices, resulted in point clouds with different orientations and shapes. Later, we'll discuss how we can determine the shape and orientation from the covariance matrix with the aid of linear algebra. What about changes to $\vec{\mu}$? Changing $\vec{\mu}$ translates the point cloud. If in the preceding examples, we had taken $\vec{\mu} = \vec{1}$ the scatterplots would have had the same shape and orientation, but they would have been tranlated by a shift of $\vec{1}$.

Definition 2.2 (Multivariate t Distribution)**.** The multivariate t-distribution on $\mathbb{R}^d$ with location vector $\vec{\mu} \in \mathbb{R}^d$, positive definite scale matrix $\mathbf{\Sigma} \in \mathbb{R}^{d \times d}$ and degrees of freedom ν has density

$$f(\vec{x}) = \frac{\Gamma\left(\frac{\nu+d}{2}\right)}{\Gamma(\nu/2)\nu^{d/2}\pi^{d/2}|\mathbf{\Sigma}|^{1/2}} \left[1 + \frac{1}{\nu}(\vec{x} - \vec{\mu})^T \mathbf{\Sigma}^{-1}(\vec{x} - \vec{\mu})\right]^{-(\nu+d)/2}.$$

We use shorthand $\vec{x} \sim t_\nu(\vec{\mu}, \mathbf{\Sigma})$ to indicate $\vec{x}$ follows this distribution.

We'll only consider a three dimensional case where the location, which determines the mode of the distribution, is $\vec{0}$ and the scale is the identity matrix. As in the Gaussian case, changing $\vec{\mu}$ translates the point cloud and different values of $\mathbf{\Sigma}$ give point clouds with different shapes. The remaining parameter to consider here is the degrees of freedom, ν, which controls how spread out the samples can be. We show results for three different choices of the degrees of freedom. For smaller degrees of freedom, there are more points which are far from the mode at $\vec{0}$.

2.3 Expectation, Mean, and Covariance

As in the one-dimensional case, the cumulative distribution function determines the distribution of the random vector, and using the density we may establish a few important quantities which will appear often throughout this text.

The first is the mean or expected value of the random vector which is the vector of expected values of each entry so that

$$E[\vec{x}] = \int_{\mathbb{R}^d} \vec{x} f(\vec{x}) d\vec{x} = \begin{bmatrix} E[x_1] \\ \vdots \\ E[x_d] \end{bmatrix} = \begin{bmatrix} \int_{\mathbb{R}^d} x_1 f(\vec{x}) d\vec{x} \\ \vdots \\ \int_{\mathbb{R}^d} x_d f(\vec{x}) d\vec{x} \end{bmatrix} \tag{2.1}$$

where

$$\int_{\mathbb{R}^d} x_i f(\vec{x}) d\vec{x} = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} x_i f(x_1, \ldots, x_d) dx_1 \ldots dx_d.$$

Note, we are assuming each of the integrals in (2.1) is well defined, which is a convention we adhere to throughout this text. Often, we'll often use $\vec{\mu}$ to denote the mean vector. When we are considering more than multiple random vectors $\vec{x}$ and $\vec{y}$ we will add a corresponding subscript $\vec{\mu}_{\vec{x}}$ to denote the corresponding mean of $\vec{x}$.

The linearity of expectation for univariate random vectors holds here as well. If $\vec{x} \in \mathbb{R}^d$ is a random vector, $\mathbf{A} \in \mathbb{R}^{k \times d}$ is a matrix of constant entries, and $\vec{b} \in \mathbb{R}^k$

is a vector of constant entries then

$$E[\mathbf{A}\vec{x} + \vec{b}] = \mathbf{A}\vec{\mu} + \vec{b}.$$

Importantly, for non-squared matrices $\mathbf{A}$ then mean of $\mathbf{A}\vec{x}$ will be of a different dimension than $\vec{x}$.

In general, the coordinates of a random vector will not be independent. To quantify the pairwise dependence, we could consider the covariance

$$Cov(x_i, x_j) = E[(x_i - \mu_i)(x_j - \mu_j)] = \int_{\mathbb{R}^d} (x_i - \mu_i)(x_j - \mu_j) f(\vec{x}) d\vec{x}$$

for $1 \leq i, j \leq d$. In the case $i = j$, this simplifies to $Cov(x_i, x_i) = Var(x_i)$.

Importantly, we do not want to consider each all of the pairwise covariance separately. Instead, we can organize them as a $d \times d$ matrix $\mathbf{\Sigma}$ with entries $\mathbf{\Sigma}_{ij} = Cov(x_i, x_j)$. Hereafter, we will refer to $\mathbf{\Sigma}$ as the covariance matrix of $\vec{x}$. When we are considering multiple random vectors we will use subscripts so that $\mathbf{\Sigma}_{\vec{x}}$ and $\mathbf{\Sigma}_{\vec{y}}$ denote the covariance matrices or random vectors $\vec{x}$ and $\vec{y}$ respectively. Following the notational conventions, it follows that $\vec{x} - E[\vec{x}] = \vec{x} - \vec{\mu} \in \mathbb{R}^d$ so that the outer product of $\vec{x} - \vec{\mu}$ with itself is the $d \times d$ matrix with entries

$$[(\vec{x} - \vec{\mu})(\vec{x} - \vec{\mu})^T]_{ij} = (x_i - \mu_i)(x_j - \mu_j)$$

so that we may more compactly write

$$\text{Var}(\vec{x}) = E\left[(\vec{x} - \vec{\mu})(\vec{x} - \vec{\mu})^T\right] \tag{2.2}$$

where we interpret the expectation operation as applying to each entry of the matrix $(\vec{x} - \vec{\mu})(\vec{x} - \vec{\mu})^T$. This looks very similar to the univariate case save that we must be mindful of the multidimensional nature of our random vector. In fact with some algebra, we have the following alternative formula for the covariance matrix

$$\mathbf{\Sigma} = E[\vec{x}\,\vec{x}^T] - \vec{\mu}\vec{\mu}^T$$

which is again reminiscent of the univariate case. Showing this result is left as a short exercise. One brief note to avoid confusion. Other texts refer to $\text{Var}(\vec{x})$ as the variance matrix or variance-covariance matrix. Herein, we use the term covariance matrix for $\text{Var}(\vec{x})$.

Recall the univariate case,

$$\text{Var}(aX + b) = a^2 \text{Var}(X)$$

for constants a and b and (one-dimensional) random variable X. Similar to the univariate case, there is a formula for the covariance of an affine mapping of a random vector, but the specific form requires us to be mindful of the matrix structure of the covariance matrix. For random vector $\vec{x} \in \mathbb{R}^d$, constant matrix $\mathbf{A} \in \mathbb{R}^{k \times d}$ and constant vector $\vec{b} \in \mathbb{R}^k$, it follows (see exercises) that

$$\mathbf{\Sigma}_{\mathbf{A}\vec{x} + \vec{b}} = \mathbf{A}\mathbf{\Sigma}\mathbf{A}^T.$$

Importantly, note that $\mathbf{A}\mathbf{\Sigma}\mathbf{A}^T$ is a $k \times k$ matrix which is consistent with the fact that $\mathbf{A}\vec{y} + \vec{b}$ is a k-dimensional vector.

Example 2.1 (Mean and Covariance of MVN)**.** If $\vec{x} \sim \mathcal{N}(\vec{\mu}, \mathbf{\Sigma})$ then $E[\vec{x}] = \vec{\mu}$ and $\text{Var}(\vec{x}) = \mathbf{\Sigma}$

Example 2.2 (Mean and Covariance of Multivariate t-distribution)**.** Let $\vec{x} \sim t_\nu(\vec{\mu}, \mathbf{\Sigma})$. If $\nu > 1$ then $E[\vec{x}] = \vec{\mu}$; otherwise the mean does not exist. If $\nu > 2$, then $\text{Var}(\vec{x}) = \frac{\nu}{\nu-2}\mathbf{\Sigma}$; otherwise, the covariance matrix does not exist.

Verifying these examples is left to the exercises and rely on multivariate change of variables which are not covered here.

2.3.1 Sample Mean and Sample Covariance

In many cases, we'll consider a collection of N *iid* vectors $\vec{x}_1, \ldots, \vec{x}_N \in \mathbb{R}$. Again, subscripts are used here, but importantly when accompanied by the $\vec{\cdot}$ sign a subscript does not refer to a specific coordinate of a vector but rather one vector in a set. Given *iid* observations $\vec{x}_1, \ldots, \vec{x}_N$, we will use sample averages to estimate the expectation and covariance of the data generating distribution. We'll use bars to denote sample averages so that $\bar{x}$ denotes the sample mean and $\bar{\mathbf{\Sigma}}$ the sample covariance. In this case, we have

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} \vec{x}_i. \tag{2.3}$$

Similarly, we define the sample covariance matrix to be

$$\bar{\mathbf{\Sigma}} = \frac{1}{N}\sum_{i=1}^{N}(\vec{x}_i - \bar{x})(\vec{x}_i - \bar{X})^T = \left(\frac{1}{N}\sum_{i=1}^{N}\vec{x}_i\vec{x}_i^T\right) - \bar{x}\bar{x}^T \tag{2.4}$$

In (2.4), dividing by N rather than $N-1$ yields biased estimates of the terms of the sample covariance matrix. However, the final formula in (2.4) more directly matches the corresponding term in the definition of the covariance matrix. Had we used a factor of $1/(N-1)$ instead, we would have

$$\frac{1}{N-1}\sum_{i=1}^{N}(\vec{x}_i - \bar{X})(\vec{x}_i - \bar{X})^T = \left(\frac{1}{N}\sum_{i=1}^{N}\vec{x}_i\vec{x}_i^T\right) - \frac{N}{N-1}\bar{x}\bar{x}^T$$

which is slightly more cumbersome. In the examples we will consider, N will typically be large enough so that the numerical difference is small. As such, we will opt for algebraically convenient definition form of (2.4) as our definition of the sample covariance matrix.

Alternatively, we can view the sample mean and sample covariance as the mean and covariance (using expectation rather than averages) of the empirical distribution from a collection of samples $\vec{x}_1, \ldots, \vec{x}_N$ defined below.

Definition 2.3 (Empirical Distribution)**.** Given a finite set of points $\mathcal{X} = \{\vec{x}_1, \ldots, \vec{x}_N\} \subset \mathbb{R}^d$, we say that random vector $\vec{z}$ follows the empirical distribution from data $\mathcal{X}$ if

$$P(\vec{z} = \vec{x}_i) = \frac{1}{N}, \quad i = 1, \ldots, N$$

and is zero otherwise.

If $\vec{z}$ follows the empirical distribution on a set of N points $\mathcal{X} = \{\vec{x}_1, \ldots, \vec{x}_N\}$, then the expectation and covariance matrix of $\vec{z}$ are equivalent to the sample mean and sample covariance for data $\vec{x}_1, \ldots, \vec{x}_N$.

2.3.2 The Data Matrix

Both (2.3) and (2.4) involve summations. Working with sums will prove cumbersome, so briefly let us introduce a more compact method for representing these expressions. Hereafter, we will organize the vectors $\vec{x}_1, \ldots, \vec{x}_N$ into a **data matrix**

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1^T \\ \vdots \\ \vec{x}_N^T \end{bmatrix} \in \mathbb{R}^{N \times d}.$$

In this setup, $\mathbf{X}_{ij}$ is the jth coordinate of $\vec{x}_i$, or equivalently, the jth measurement taken from the ith subject. Thus, rows of $\mathbf{X}$ index subjects (realizations of the random vector) whereas columns index common measurements across all subjects. Using the data matrix, we can forgo the summation notation giving the following formulas for the sample mean

$$\bar{x} = \frac{1}{N}\mathbf{X}^T\vec{1} \tag{2.5}$$

and the sample covariace matrix

$$\bar{\mathbf{\Sigma}} = \frac{1}{N}(\mathbf{HX})^T\mathbf{HX} = \frac{1}{N}\mathbf{X}^T\mathbf{HX} \tag{2.6}$$

where $\mathbf{H} = \mathbf{I} - \frac{1}{N}\vec{1}\vec{1}^T \in \mathbb{R}^{N \times N}$ is known as the centering matrix. We have used the fact that $\mathbf{H}$ is symmetric and idempotent, e.g. $\mathbf{H}^2 = \mathbf{H}$ which is left as a exercise. The vector $\vec{1}$ is the N-dimensional vector with 1 in each entry and $\mathbf{I}$ is the $N \times N$ identity matrix. One can show (see exercises) that the matrix-vector and matrix-matrix multiplication implicitly handles the summations in (2.3) and (2.4).

To conclude this section, we compare the sample mean and covariance matrix computed from random draws from the MVN distribution.

Example 2.3 (Draws from the MVN)**.** We draw $N = 100$ samples from the $\mathcal{N}(\vec{0}, \mathbf{\Sigma})$ distribution where

$$\mathbf{\Sigma} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 9 \end{bmatrix}$$

.

```
N <- 100
X <- mvrnorm(n=N, mu = rep(0,3), Sigma = c(1,4,9)*diag(3))
```

To compute the sample mean and sample covariance, we implement (2.3) and (2.4).

```
xbar <- (1/N) * t(X) %*% rep(1,N)
H <- diag(N) - (1/N)*matrix(1,nrow = N, ncol = N)
S <- (1/N) *  t(H %*% X) %*% (H %*% X)
```

The results are shown below (rounded to three decimal places)

$$\bar{x} = \begin{bmatrix} 0.008 \\ 0.078 \\ -0.239 \end{bmatrix} \quad \text{and} \quad \bar{\mathbf{\Sigma}} = \begin{bmatrix} 1.148 & 0.216 & 0.121 \\ 0.216 & 4.181 & 0.087 \\ 0.121 & 0.087 & 9.366 \end{bmatrix}$$

which are close to the true values. If we increase the sample size to $N = 10^4$ samples, we get estimates which are closer to the true values (shown below).

$$\bar{x} = \begin{bmatrix} 0.001 \\ 0.02 \\ 0.051 \end{bmatrix} \quad \text{and} \quad \bar{\mathbf{\Sigma}} = \begin{bmatrix} 0.991 & -0.02 & -0.04 \\ -0.02 & 3.997 & 0.095 \\ -0.04 & 0.095 & 8.992 \end{bmatrix}$$

2.4 Linear Algebra

2.4.1 Assumed Background

This text assumes familiarity with definitions from a standard undergraduate course in linear algebra including but not limited to linear spaces, subspaces, spans and bases, and matrix multiplication. However, we have elected to provide review of some of the most commonly used ideas in the methods we'll cover in the following subsections. For a more thorough treatment of linear algebra, please see [**?**, **?**]

2.4.2 Interpretations of Matrix Multiplication

Throughout this text, comfort with common calculations in linear algebra will be very important. Herein, we assume the reader has some exposure to these materials at an undergraduate level including the summation of vectors or matrices. The familiarity with matrix-vector and matrix-matrix multiplication will play a central role as we have already seen in the case of the data matrix formulation of the sample mean and sample covariance. However, rote familiarity with computation will not be sufficient to build intuition for the methods we'll discuss. As such, we'll begin with a review of a few important ways one can view matrix-vector (and matrix-matrix) multiplication which will be helpful

later. Those who feel comfortable with the myriad interpretations of matrix multiplication in terms of linear combinations of the rows and columns may skip to the next section.

Suppose we have matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ and vector $\vec{x} \in \mathbb{R}^n$. If we let $\vec{a}_1^T, \ldots, \vec{a}_M^T \in \mathbb{R}^n$ denote the rows of $\mathbf{A}$, then the most commonly cited formula for computing $\mathbf{A}\vec{x}$ is

$$\mathbf{A}\vec{x} = \begin{bmatrix} \vec{a}_1^T \\ \vdots \\ \vec{a}_m^T \end{bmatrix} \vec{x} = \begin{bmatrix} \vec{a}_1^T\vec{x} \\ \vdots \\ \vec{a}_m^T\vec{x} \end{bmatrix}$$

wherein we take the inner product of the rows of $\mathbf{A}$ with vector $\vec{x}$. We can expand this definition to matrix-matrix multiplication. If $\mathbf{B} \in \mathbb{R}^{n\times k}$ has columns $\vec{b}_1, \ldots, \vec{b}_k$ then

$$(\mathbf{AB})_{ij} = \vec{a}_i^T\vec{b}_j$$

where we take the inner product of the ith row of $\mathbf{A}$ with the jth column of $\mathbf{B}$ to get the ijth entry of $\mathbf{AB}$. This is perfectly reasonable method of computation, but alternative perspectives are helpful, particularly when we consider different factorization of the data matrix in later chapters..

Returning to $\mathbf{A}\vec{x}$, suppose now that $\mathbf{A}$ has columns $\vec{\alpha}_1, \ldots, \vec{\alpha}_n$ and $\vec{x} = (x_1, \ldots, x_n)^T$, then we may view $\mathbf{A}\vec{x}$ as a linear combination of the columns of $\mathbf{A}$ so that

$$\mathbf{A}\vec{x} = \begin{bmatrix} \vec{\alpha}_1 \,| & \cdots & |\, \vec{\alpha}_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = x_1\vec{\alpha}_1 + \cdots + x_n\vec{\alpha}_n = \sum_{j=1}^{n} x_j\vec{\alpha}_j.$$

Here, we added vertical columns between the vectors $\vec{\alpha}_i$ to make clear that $\begin{bmatrix} \vec{\alpha}_1 \,| & \cdots & |\, \vec{\alpha}_n \end{bmatrix}$ is a matrix. We can extend this perspective to see that the columns of $\mathbf{AB}$ are comprised of different linear combinations of the columns of $\mathbf{A}$. Specifically, the j column of $\mathbf{AB}$ is a linear combination of the columns of $\mathbf{A}$ using the entries in the jth column of $\mathbf{B}$. More specifically, the jth column of $\mathbf{AB}$ is the linear combination

$$\sum_{i=1}^{n} \mathbf{B}_{ij}\vec{\alpha}_i.$$

Our final observations follows by taking these insights on linear combinations of columns and transposing the entire operation. What can we say about the rows of $\mathbf{AB}$? We can rewrite $\mathbf{AB} = (\mathbf{B}^T\mathbf{A}^T)^T$. The columns of $\mathbf{B}^T\mathbf{A}^T$ are linear combinations of the columns of $\mathbf{B}^T$. Since the columns of $\mathbf{B}^T$ are the rows of $\mathbf{B}$, it follows that the rows of $\mathbf{AB} = (\mathbf{B}^T\mathbf{A}^T)^T$ are linear combinations of the rows of $\mathbf{B}$ with weights given by the entries in each row of $\mathbf{A}$ respectively. In mathematical notation, the ith row of $\mathbf{AB}$ is

$$\sum_{j=1}^{n} \mathbf{A}_{ij}\vec{\beta}_j^T$$

where $\vec{\beta}_1^T, \ldots, \vec{\beta}_n^T$ are the rows of $\mathbf{B}$.

2.4.3 Norms and Distances

Throughout this text, we will use $\|\cdot\|$ to denote the usual Euclidean (or ℓ_2) norm, which for a vector, $\vec{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$, is

$$\|\vec{x}\| = \left(\sum_{j=1}^{d} x_j^2 \right)^{1/2}.$$

We may then define the Euclidean distance between two d-dimension vectors $\vec{x}$ and $\vec{y}$ to be

$$\|\vec{x} - \vec{y}\| = \left(\sum_{j=1}^{d} (x_j - y_j)^2 \right)^{1/2}.$$

Euclidean distance is the most commonly used notion of distance (or norm or metric) between two vectors, but it is far from the only option. We can consider the general ℓ_p norm

$$\|\vec{x}\|_p = \left(\sum_{j=1}^{d} x_j^p \right)^{1/p}$$

which coincides with the Euclidean norm for $p = 2$. Two other special cases include $p = 1$ also known as the Manhattan distance and $p = \infty$ also known as the sup-norm

$$\|\vec{x}\|_\infty = \max_{j=1,\ldots,d} |x_j|.$$

We can also extend this notions of vector norms to a measure of the norm of a matrix. Two important cases are the ℓ_2 norm of a matrix and the Frobenius norm. For matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the ℓ_2 norm is

$$\|\mathbf{A}\| = \sup_{\vec{x} \in \mathbb{R}^n \text{ s.t. } \vec{x} \neq \vec{0}} \frac{\|\mathbf{A}\vec{x}\|}{\|\vec{x}\|} = \sup_{\vec{x} \in \mathbb{R}^n \text{ s.t. } \|\vec{x}\| = 1} \|\mathbf{A}\vec{x}\|. \tag{2.7}$$

You can interpret $\|\mathbf{A}\|$ as the largest relative change in the Euclidean length of a vector after it is multiplied by $\mathbf{A}$. The Frobenius norm extends the algebraic definition notion of Euclidean length to a matrix. For $\mathbf{A} \in \mathbb{R}^{m \times n}$, its Frobenius norm is

$$\|\mathbf{A}\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{A}_{ij}^2 \right)^{1/2}. \tag{2.8}$$

The ℓ_2 distance between two matrices is then $\|\mathbf{A} - \mathbf{B}\|$ and the Frobenius distance between two matrices is $\|\mathbf{A} - \mathbf{B}\|_F$ (where both $\mathbf{A}$ and $\mathbf{B}$ have the same number of rows and columns).

2.4.4 Important properties

A few additional definitions that we will use throughout the text are provided below without examples.

Definition 2.4 (Symmetric Matric)**.** Matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$ is symmetric if $\mathbf{A} = \mathbf{A}^T$.

Definition 2.5 (Eigenvectors and Eigenvalues)**.** Let $\mathbf{A} \in \mathbb{R}^{d \times d}$. If there is a scalar λ and vector $\vec{x} \neq \vec{0}$ such that $\mathbf{A}\vec{x} = \lambda\vec{x}$ then we say λ is an eigenvalue of $\mathbf{A}$ with associated eigenvector $\vec{x}$.

2.4.5 Matrix Factorizations

Two different matrix factorization will arise many times throughout the text. The first, which is commonly presented in linear algebra courses, is the spectral decomposition of a square matrix which is also known as diagonalization or eigenvalue decomposition. Herein, we assume familiarity with eigenvalues and eigenvectors. The second factorization is the singular value decomposition. In the subsequent subsections, we briefly discuss these two factorizations, their geometric interpretation, and some notation that will typically be used in each case.

2.4.5.1 Eigenvalue Decomposition

We begin with the eigenvalue decomposition of a square matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$. As you may recall, $\mathbf{A}$ will have a set of d eigenvalues $\lambda_1, \ldots, \lambda_d$ (which may include repeated values) and associated eigenvectors. A number, λ, may be repeated in the list of eigenvalues, and the number of times is called the algebraic multiplicity of λ. Each eigenvalue has a least one eigenvector. In cases where the eigenvalue has algebraic multiplicity greater than one, we refer to its geometric multiplicity as the number of linearly independent eigenvectors associated with the eigenvalue.

The algebraic multiplicity is always greater than or equal to the geometric multiplicity. However, this is not always the case, and when this occurs, the matrix cannot be diagonalized. Fortunately, we will largely be dealing with symmetric matrices for which diagonalization is guaranteed by the following theorem.

Theorem 2.1 (Spectral Decomposition Theorem for Symmetric Matrices)**.** *Any symmetric matrix* $\mathbf{A} \in \mathbb{R}^{d \times d}$ *can be written as*

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$$

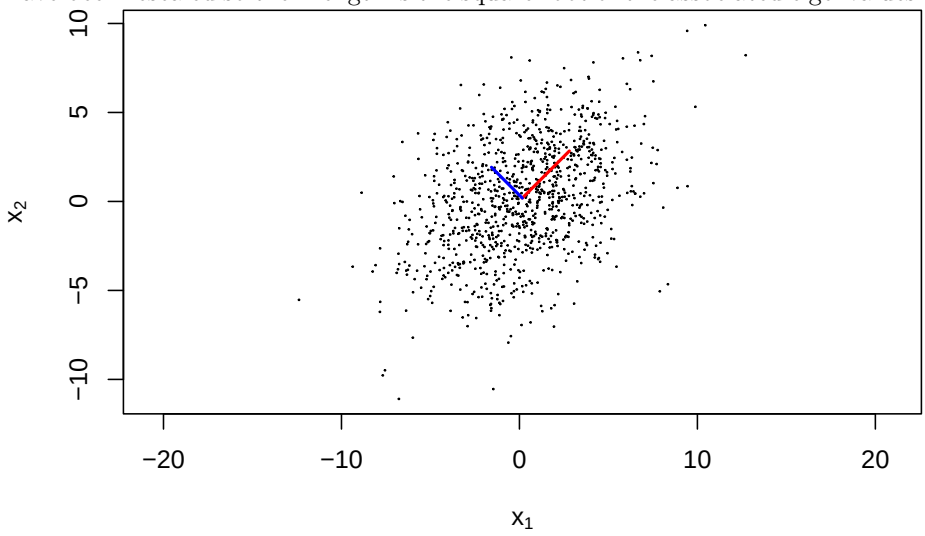
where $\mathbf{U} \in \mathbb{R}^{d \times d}$ *is an orthonormal matrix and* $\mathbf{\Lambda}$ *is a diagonal matrix*

$$\mathbf{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_d \end{bmatrix}$$

where the scalars $\lambda_1, \ldots, \lambda_d \in \mathbb{R}$ *are the eigenvalues of* $\mathbf{A}$ *and the corresponding columns of* $\mathbf{U}$ *are their associated eigenvectors.*

By convention, we will always assume the eigenvalues are in decreasing order so that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$. The most common types of symmetric matrices that we will encounter are covariance matrices. In those cases, the spectral decomposition of the covariance can provide some helpful insight about the shape generated by many *iid* samples from a distribution. We demonstrate this idea graphically in the following examples using the MVN.

Example 2.4 (Level curves of MVN in)**.** A scatterplot of 1000 *iid* samples from two-dimensional Gaussian distribution with mean $\vec{0}$ and covariance $\begin{bmatrix} 10 & 2 \\ 2 & 10 \end{bmatrix}$ is shown below. The eigenvector associated with the largest eigenvalue of the sample covariance matrix is shown in red, whereas the eigenvector shown associated with the second eigenvalue is shown in blue. Note, both eigenvectors have been rescaled so their length is the square root of the associated eigenvalues.



Importantly, observe how these eigenvectors follow the spread of the data. As we will see in 4.1, this idea can be used to compress data in a geometrically interpretable manner through PCA.

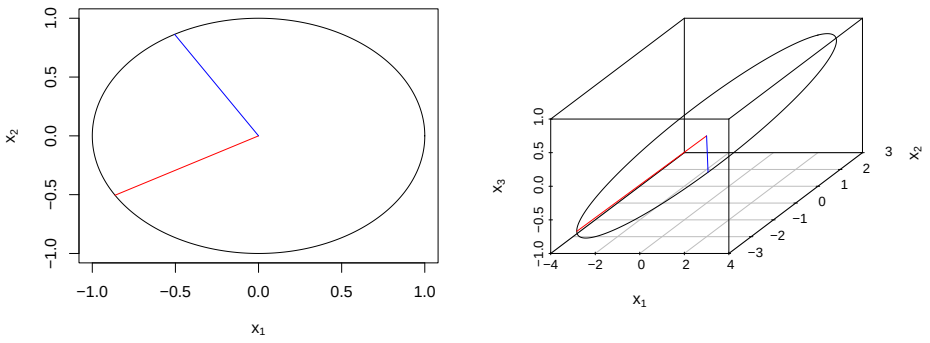
2.4.5.2 Singular Value Decomposition

The spectral theorem is limited in that it requires a matrix to be both square and symmetric. When focusing on data matrices $\mathbf{X} \in \mathbb{R}^{N \times d}$ both assumptions are extremely unlikely to be satisfied, and we will need a more flexible class of methods. This idea is explored in much greater detail in Chapter 4. For now, we briefly introduce the Singular Value Decomposition and how this factorization provides some insight on the geometric structure of matrix-vector multiplication.

For a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ the mapping $T_{\mathbf{A}} : \mathbb{R}^n \mapsto \mathbb{R}^m$ defined as $T_{\mathbf{A}}(\vec{x}) = \mathbf{A}\vec{x}$ for $\vec{x} \in \mathbb{R}^n$. Since this mapping is linear, we can understand its geometric nature by investigating how the unit sphere in $\mathbb{R}^n$ is transformed by the mapping $T_{\mathbf{A}}$. More directly, let's suppose we took every vector on the unit sphere $\mathcal{S} = \{\vec{x} \in \mathbb{R}^n : \|\vec{x}\| = 1\}$ and computed the vector $\mathbf{A}\vec{x}$. The result is a hyperellipse in $\mathbb{R}^m$. The specifics on the orientation and shape of the hyperellipse depend on the matrix $\mathbf{A}$ which we demonstrate through a brief example.

Example 2.5. In Figure **??**, we show the unit sphere in $\mathbb{R}^2$ and the resulting hyperellipse in $\mathbb{R}^3$ after multiplication by the matrix

$$\mathbf{A} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \\ 1 & 0 \end{bmatrix}.$$



We have also added two vectors. On the hyperellipse, the red vector is the longest semi-axis of the ellipse; the blue vector is the shortest semi-axis. The red and blue vectors on the left are the preimages of the semi-axes of the ellipse (those vectors which get mapped to the semi-axes after multiplication with $\mathbf{A}$). It may not be clear from the static images above, but the semi-major axes on the right are orthogonal. We provide an interactive plot of hyperellipse below so that you can better see this result. Interestingly, the preimages on the left are also perpendicular. Since they are on the unit sphere, they are also orthonormal.

In the preciding example, we have made a few important observations which generalize to any multiplication by a matrix, $\mathbf{A} \in \mathbb{R}^{m \times n}$. For now, assume that $m > n$ and $\mathbf{A}$ is full rank. In this case, the hyperellipse $\mathbf{A}\mathcal{S} = \{\mathbf{A}\vec{x} : \vec{x} \in \mathcal{S}\}$ has orthogonal semi-axes $\sigma_1\vec{u}_1, \ldots, \sigma_n\vec{u}_n$ with corresponding orthonormal preimages $\vec{v}_1, \ldots, \vec{v}_n$. We use the assumption that the lengths of the axes are listed in decreasing order so that $\sigma_1 \geq \cdots \geq \sigma_n$. We may express these mappings in the following manner

$$\mathbf{A} \underbrace{\begin{bmatrix} | & & | \\ \vec{v}_1 & \ldots & \vec{v}_n \\ | & & | \end{bmatrix}}_{\tilde{\mathbf{V}}} = \begin{bmatrix} | & & | \\ \sigma_1\vec{u}_1 & \ldots & \sigma_n\vec{u}_n \\ | & & | \end{bmatrix} = \underbrace{\begin{bmatrix} | & & | \\ \vec{u}_1 & \ldots & \vec{u}_n \\ | & & | \end{bmatrix}}_{\tilde{\mathbf{U}}} \underbrace{\begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_n \end{bmatrix}}_{\tilde{\mathbf{S}}} \tag{2.9}$$

Since $\tilde{\mathbf{V}} \in \mathbb{R}^{n \times n}$ has orthonormal columns so that $\tilde{\mathbf{V}}^{-1} = \tilde{\mathbf{V}}^T$, we arrive at the following decomposition for $\mathbf{A}$.

Definition 2.6 (The Singular Value Decomposition (SVD))**.** Let $\mathbf{A} \in \mathbb{R}^{m \times n}$. Then $\mathbf{A}$ may be factored as

$$\mathbf{A} = \tilde{\mathbf{U}}\tilde{\mathbf{S}}\tilde{\mathbf{V}}^T \tag{2.10}$$

where $\tilde{\mathbf{U}} \in \mathbb{R}^{m \times n}$ and $\tilde{\mathbf{V}} \in \mathbb{R}^{n \times n}$ have orthonormal columns and $\tilde{\mathbf{S}} \in \mathbb{R}^{n \times n}$ is a diagonal matrix with real entries $\sigma_1 \geq \ldots \sigma_n \geq 0$ along the diagonal. We refer to (2.10) as the reduced singular value decomposition of $\mathbf{A}$. The columns of $\tilde{\mathbf{U}}$ ($\tilde{\mathbf{V}}$)

are called the left (right) singular vectors of $\mathbf{A}$ and the scalars $\sigma_1 \geq \cdots \geq \sigma_n$ are referred to as the singular values of $\mathbf{A}$.

Let $\vec{u}_1, \ldots, \vec{u}_n \in \mathbb{R}^m$ be the columns of $\tilde{\mathbf{U}}$. When $m > n$, we may find additional vectors $\vec{u}_{n+1}, \ldots, \vec{u}_m$ such that $\{\vec{u}_1, \ldots, \vec{u}_m\}$ are an orthonormal basis for $\mathbb{R}^m$. This gives the (full) singular value decomposition of $\mathbf{A}$ is

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T \tag{2.11}$$

where

$$\mathbf{U} = \begin{bmatrix} \vec{u}_1 & \ldots & \vec{u}_m \end{bmatrix} \in \mathbb{R}^{m \times m} \qquad \mathbf{V} = \begin{bmatrix} \vec{v}_1 & \ldots & \vec{v}_n \end{bmatrix} \in \mathbb{R}^{n \times n}$$

are orthonormal matrices. The matrix $\mathbf{S} \in \mathbb{R}^{m \times n}$ is formed by taking $\tilde{S}$ and padding it with zero along the bottom and right so that it is $m \times n$.

We may drop the preceding assumptions that $m > n$ or the $\mathbf{A}$ is full rank. In fact, **every** matrix has a singular value decomposition. A proof of this fact using induction may be found in [**?**]. Furthermore, a few important matrix properties are directly related to the SVD.

1) The rank of a matrix is equal to its number of positive singular values
2) The column space of a matrix is the span of its first $rank(\mathbf{A})$ left singular vectors
3) The kernel of a matrix is the span of its last $n - rank(\mathbf{A})$ right singular vectors.

When considering matrix muliplication $\mathbf{A}\vec{x}$, the full SVD of $\mathbf{A}$ is helpful for decomposition this matrix multiplication into three more interpretable steps: a rotation/reflection, a stretch/compression, and a final rotation/reflection.

$$\mathbf{A}\vec{x} = \underbrace{\mathbf{U}}_{rot/ref}\underbrace{\underbrace{\mathbf{D}\underbrace{(\mathbf{V}^T\vec{x})}_{rot/ref}}_{str/com}}_{rot/ref} \tag{2.12}$$

2.4.6 Positive Definiteness and Matrix Powers

Finally, we review a few brief properties for square matrices.

Definition 2.7 (Positive Definite Matrices)**.** A matrix, $\mathbf{A} \in \mathbb{R}^{n \times n}$ is positive definite if $\vec{x}^T\mathbf{A}\vec{x} > 0$ for all $\vec{x} \neq 0$. Equivalently, $\mathbf{A}$ is positive definite if all of its eigenvalues are positive.

A matrix is positive semidefinite if $\vec{x}^T\mathbf{A}\vec{x} \geq 0$ for all $\vec{x} \neq 0$. Equivalently, $\mathbf{A}$ is positive semidefinite if all of its eigenvalues are non-negative.

One may interpret the powers of a matrix as repeated multiplication of a square matrix. More specifically, for any $m \in \{1, 2, \dots\}$, let $\mathbf{A}^m = \underbrace{\mathbf{A} \dots \mathbf{A}}_{m \text{ times}}$. For diagonalizable positive definite matrices, we may extend this interpretation to negative and fractional power using the eigenvalue decomposition. If the eigenvalue decomposition of the $\mathbf{A}$ is $\mathbf{A} = \mathbf{W}\mathbf{\Lambda}\mathbf{W}^{-1}$ for some diagonal matrix Λ with positive entries, $\lambda_1, \dots, \lambda_n$ along its diagonal. For any $s \in \mathbb{R}$, we define $\mathbf{A}^s = \mathbf{W}\mathbf{\Lambda}^s\mathbf{W}^{-1}$ where

$$\Lambda^s = \begin{bmatrix} \lambda_1^s & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n^s \end{bmatrix}.$$

2.4.7 Hadamard (elementwise) operations

Given two matrices $\mathbf{A}$, $\mathbf{B} \in \mathbb{R}^{m \times n}$ with the same number of rows and columns, we define Hadamard product and division as

$$\mathbf{A} \odot \mathbf{B} \in \mathbb{R}^{m \times n} \text{ such that } (\mathbf{A} \odot \mathbf{B})_{ij} = \mathbf{A}_{ij}\mathbf{B}_{ij}$$
$$\mathbf{A} \oslash \mathbf{B} \in \mathbb{R}^{m \times n} \text{ such that } (\mathbf{A} \oslash \mathbf{B})_{ij} = \frac{\mathbf{A}_{ij}}{\mathbf{B}_{ij}}$$

Unlike standard matrix multiplication, the Hadamard product $\odot$ is commutative such that $\mathbf{B} \odot \mathbf{A} = \mathbf{A} \odot \mathbf{B}$. Commutativity does not hold for Hadamard division however.

Furthermore, we define Hadamard powers as

$$\mathbf{A}^{\circ k} \in \mathbb{R}^{m \times n} \text{ such that } \left(\mathbf{A}^{\circ k}\right)_{ij} = \left(\mathbf{A}_{ij}\right)^k.$$

For the special case $k = -1$, we refer to $\mathbf{A}^{\circ(-1)}$ as the Hadamard inverse.

2.5 Exercises

2.5.1 Probability

1. Given a random vector $\vec{x} = (\mathbf{x}_1, \mathbf{x}_2)^T \sim \mathcal{N}(\vec{\mu}, \mathbf{\Sigma})$ where:

$$\vec{\mu} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \mathbf{\Sigma} = \begin{pmatrix} 1 & 0 \\ 0 & 10 \end{pmatrix}$$

   a. What are the means and variances of the individual components x_1 and x_2?

   b. What is the covariance and correlation between x_1 and x_2?

   c. Are $\mathbf{x}_1$ and $\mathbf{x}_2$ independent?

2. Given a random vector $\vec{x} = (x_1, x_2)^T \sim \mathcal{N}(\vec{\mu}, \mathbf{\Sigma})$ where:

$$\vec{\mu} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad \mathbf{\Sigma} = \begin{pmatrix} 4 & 2 \\ 2 & 3 \end{pmatrix}$$

   a. What are the means and variances of the individual components $\mathbf{x}_1$ and $\mathbf{x}_2$?

   b. What is the covariance and correlation between $\mathbf{x}_1$ and $\mathbf{x}_2$?

   c. Are $\mathbf{x}_1$ and $\mathbf{x}_2$ independent?

3. Given a random vector $\vec{x} = (x_1, x_2) \sim t_{10}(\vec{\mu}, \mathbf{\Sigma})$ where:

$$\vec{\mu} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \mathbf{\Sigma} = \begin{pmatrix} 1 & 0 \\ 0 & 10 \end{pmatrix}$$

   a. What are the means and variances of the individual components $\mathbf{x}_1$ and $\mathbf{x}_2$?

   b. What is the covariance and correlation between $\mathbf{x}_1$ and $\mathbf{x}_2$?

   c. Are $\mathbf{x}_1$ and $\mathbf{x}_2$ independent?

4. Given a random vector $\vec{x} = (x_1, x_2) \sim t_{10}(\vec{\mu}, \mathbf{\Sigma})$ where:

$$\vec{\mu} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad \mathbf{\Sigma} = \begin{pmatrix} 4 & 2 \\ 2 & 3 \end{pmatrix}$$

   a. What are the means and variances of the individual components $\mathbf{x}_1$ and $\mathbf{x}_2$?

   b. What is the covariance and correlation between $\mathbf{x}_1$ and $\mathbf{x}_2$?

   c. Are $\mathbf{x}_1$ and $\mathbf{x}_2$ independent?

2.5.2 Calculus

1. Let $f(\vec{x}) = \vec{x}^T \mathbf{A} \vec{x}$ for a vector $\vec{x} \in \mathbb{R}^d$ and a matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$ which is constant. Give expressions for ∇f and $\mathcal{H} f$ using only matrices and vectors (no summation notation is allowed).

2. Let $\vec{x} \in \mathbb{R}^m$ and $\vec{y} \in \mathbb{R}^n$. Compute the gradient of $\|\vec{x}\vec{y}^T\|_F^2$ with respect to $\vec{x}$.

3. Consider the function

$$f(x_1, x_2, x_3) = 2x_1^2 + 3x_2^2 + 4x_3^2 + 5x_1x_2 + 6x_1x_3 + 7x_2x_3 + 8x_1 + 9x_2 + 10x_3 + 11$$

   a. Find the critical point(s) of f. Hint: it may be helpful to express f in the form $f(\vec{x}) = \vec{x}^T \mathbf{A} \vec{x} + \vec{b}^T \vec{x} + c$ for some matrix $\mathbf{A}$, vectors $\vec{b}, \vec{x} = (x_1, x_2, x_3)^T \in \mathbb{R}^3$ and scalar c.

b. Classify the critical point(s) found in part (a) as minima, maxima, or saddle points. Explain your reasoning.

4. Verify that the mean and variance of the $\mathcal{N}(\vec{\mu}, \mathbf{\Sigma})$ distribution are indeed $\vec{\mu}$, $\mathbf{\Sigma}$ respectively by computing the integrals directly.

5. Verify that the mean and variance of the $t_\nu(\vec{\mu}, \mathbf{\Sigma}$ distribution by computing the integrals directly. You may assume that $\nu > 2$.

2.5.3 Linear Algebra

1. Given a matrix $\mathbf{A} \in \mathbb{R}^{3\times 3}$ and a vector $\mathbf{x} \in \mathbb{R}^3$, where:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$$

   a. Compute the product $\mathbf{Ax}$ using the column interpretation of matrix-vector multiplication.

   b. Compute the product $\mathbf{Ax}$ using the row interpretation of matrix-vector multiplication.

2. Given matrices $\mathbf{A} \in \mathbb{R}^{2\times 3}$ and $\mathbf{B} \in \mathbb{R}^{3\times 2}$, where:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

   a. Compute the product $\mathbf{C} = \mathbf{AB}$ using the dot product interpretation of matrix-matrix multiplication.

   b. Compute the product $\mathbf{C} = \mathbf{AB}$ using the column interpretation of matrix-matrix multiplication.

   c. Compute the product $\mathbf{C} = \mathbf{AB}$ using the row interpretation of matrix-matrix multiplication.

3. Let $\mathbf{A} \in \mathbb{R}^{m\times n}$.

   a. Explain why $\mathbf{AA}^T$ and $\mathbf{A}^T\mathbf{A}$ are diagonalizable and positive semi-definite.

   b. Give expressions for the singular vectors and singular values of $\mathbf{A}$ in terms of the eigenvectors and eigenvalues of $\mathbf{AA}^T$ and $\mathbf{A}^T\mathbf{A}$.

4. Let

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

   a. Show that $\mathbf{A}^2 \neq \mathbf{A}^{\circ 2}$.

   b. Under what conditions does $\mathbf{A}^k = \mathbf{A}^{\circ k}$ for a square matrix $\mathbf{A}$?

2.5.4 Hybrid Problems

1. Consider the data matrix

$$\mathbf{X} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix}.$$

   a. Compute the sample mean of the data.
   b. Provide the centered data matrix.
   c. Compute the sample covariance matrix of $\mathbf{X}$.

2. Consider a random vector $\mathbf{X} \sim \mathcal{N}(\mu, \mathbf{\Sigma})$ where:

$$\mu = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \mathbf{\Sigma} = \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix}.$$

   Let $\mathbf{Y} = \mathbf{A}\mathbf{X} + \mathbf{b}$ where:

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

   a. What is the mean vector $\mu_{\mathbf{Y}}$ of $\mathbf{Y}$?

   b. What is the covariance matrix $\mathbf{\Sigma}_{\mathbf{Y}}$ of $\mathbf{Y}$?

3. Given a random vector $\vec{x}$ with mean $\vec{\mu}$ and covariance matrix $\mathbf{\Sigma} = E[(\vec{x} - \vec{\mu})(\vec{x} - \vec{\mu})^T]$ verify the identity $\mathbf{\Sigma} = E[\,\vec{x}\,\vec{x}^T\,] - \vec{\mu}\,\vec{\mu}^T$.

4. Suppose $\vec{x} \in \mathbb{R}^n$ is a random vector with covariance $\mathbf{\Sigma} \in \mathbb{R}^{n \times n}$, and let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a matrix with constant entries (non-random). Show that the covariance matrix of $\mathbf{A}\vec{x}$ is $\mathbf{A}\mathbf{\Sigma}\mathbf{A}^T$.

5. Show that the $N \times N$ centering matrix $\mathbf{H} = \mathbf{I} - \frac{1}{N}\mathbb{1}\mathbb{1}^T$ is idempotent, i.e. $\mathbf{H}^2 = \mathbf{H}$.

6. Consider the data matrix

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1^T \\ \vdots \\ \vec{x}_N^T \end{bmatrix}.$$

   Show that

$$\mathbf{H}\mathbf{X} = \begin{bmatrix} \vec{x}_1^T - \bar{x}^T \\ \vdots \\ \vec{x}_N^T - \bar{x}^T \end{bmatrix}$$

   where $\mathbf{H}$ is the centering matrix and $\bar{x}$ is the sample mean of vectors $\vec{x}_1, \ldots, \vec{x}_N$.

7. For data $\vec{x}_1, \ldots, \vec{x}_N$ with sample mean

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} \vec{x}_i$$

and sample covariance

$$\hat{\mathbf{\Sigma}} = \frac{1}{N} (\vec{x}_i - \hat{\mu})(\vec{x}_i - \hat{\mu})^T$$

verify the following identities

   a. $\hat{\mathbf{\Sigma}} = \frac{1}{N} \left(\sum_{i=1}^{N} \vec{x}_i \vec{x}_i^T \right) - \hat{\mu}\hat{\mu}^T$.

   b. $\hat{\mathbf{\Sigma}} = \mathbf{X}^T \mathbf{H} \mathbf{X}$ where $\mathbf{X}$ is the data matrix formed from vectors $\vec{x}_1, \ldots, \vec{x}_N$.

Chapter 3

Central goals and assumptions

In the remainder of this text, we will largely focus on the case where we are given a dataset containing samples $\vec{x}_1, \ldots, \vec{x}_N \in \mathbb{R}^d$. We will assume that the vectors were drawn independently from some unknown data generating process. As we discussed briefly in Chapter 1, in UL we want to learn important relationships within the dataset that can provide a simplified but meaningful summary of the data. The central assumption to UL is that such structure exists, though the specifics vary depending on the setting.

The two largest areas of focus herein are dimension reduction/manifold learning and clustering which can both be used for feature engineering, data compression, and exploratory data analysis. Dimension reduction is also commonly used for visualization. We'll briefly discuss association rules in Section 4.2.

3.1 Dimension reduction and manifold learning

Algorithms for dimension reduction and manifold learning have a number of different applications which are all based on the Manifold Hypothesis

Hypothesis 3.1 (Manifold Hypothesis)**.** The points in a high-dimensional dataset live on a latent low-dimension surface (also called a manifold).

The manifold hypothesis implies that the dataset can be described by a much smaller number of dimensions. Determining the manifold structure and a simplified set of coordinates for each point on the manifold is the central goal of these algorithms.

3.2 Clustering

Hypothesis 3.2 (Clustering Hypothesis)**.** The points in a dataset can be grouped into well defined subsets. Points in each subset are similar and points in different subsets are not.

The cluster hypothesis suggests there are non-overlapping regions. Within each region there are (many) similar points, and there are no points whose are comparably similar to those in more than one subset.

3.3 Generating synthetic data

3.3.1 Data on manifolds

At the beginning of this chapter, we indicated that our data are drawn from some unknown distribution. This is a practical assumption, but in many cases, it is also helpful to consider examples where we generate the data ourselves. In doing so, we can create whatever complicated structure we would like such as different clustering arrangements or lower dimensional structure. We can test an Unsupervised Learning algorithm of interest on these synthetically generated data to see if important relationships or properties are accurately preserved. This is a helpful method for evaluating how well an algorithm works in a specific case, and importantly, can be used to build intuition on a number of natural complexities such as appropriately choosing tuning parameters, evaluating the effects of noise, and seeing how these algorithms may break when certain assumptions are not met.

First, let us consider the case of generating data with a known lower dimensional structure which will be valuable when testing a dimension reduction or manifold learning algorithm. We'll begin with data on a hyperplane. Later in Chapter 4, we consider data on a hyperplane with additional constraints which can be generated by small changes to the method discussed below.

Example 3.1 (Generating data on a hyperplane)**.** Suppose we want to generate a set of d-dimensional data which is on a $k < d$ dimensional hyperplane. The span of k linearly independent vectors $\vec{z}_1, \ldots, \vec{z}_k \in \mathbb{R}^d$ defines a k dimensional hyperplane. If we then generated random coefficient $c_1, \ldots, c_k$, then the vector

$$\vec{x} = c_1\vec{z}_1 + \cdots + c_k\vec{z}_k$$

would be an element on this hyperplane. To generate a data set we could then

1) Specify $\vec{z}_1, \ldots, \vec{z}_k$ or generate them randomly
2) Draw random coefficients $c_1, \ldots, c_k$ and compute the random sample $\vec{x} = c_1\vec{z}_1 + \cdots + c_k\vec{z}_k$.
3) Repeat step 2, N times to generate the N samples.

In Figure 3.1, we show an example of data generated to reside on a random plane in $\mathbb{R}^3$. We first generate $\vec{z}_1, \ldots, \vec{z}_k$ randomly by drawing each vector from a $\mathcal{N}(\vec{0}, \mathbf{I})$ distribution. These vectors will be independent with probability 1. When then take coefficients $c_1, \ldots, c_k$ which are **iid** $N(0, 1)$.

```
set.seed(185)
N <- 100
# basis
zs <- mvrnorm(n=2, mu = rep(0,3), Sigma = diag(1,3))
#coeffs
coeffs <- matrix(rnorm(2*N),ncol = 2)
# generate data matrix
samples <- coeffs %*% zs
# plot results
scatterplot3js(samples,
               xlab = expression(x[1]), ylab = expression(x[2]), zlab = expression(x[3]),
               angle = 90,
               pch = '.',
               size = 0.2)
```

Figure 3.1: Randomly generated points concentrated on a two-dimensional hyperplane.

Generating data on a curved surface is generally more complicated. In some cases, the curved surface is defined implicitly via a constraint such as the unit

sphere in d-dimensions

$$S^{d-1} = \{\vec{x} \in \mathbb{R}^d : \|\vec{x}\| = 1\}.$$

Generating data on the unit sphere can then be accomplished by drawing a vector from any distribution on $\mathbb{R}^d$ then rescaling the vector to have unit length. Different choices of the original distribution will result in different distributions over the unit sphere.

Alternatively, we could consider a function which parameterizes a curve or surface. We show one such example below.

Example 3.2 (Generating data on a Ribbon)**.** The function

$$R(s) = (\cos(2\pi s), \sin(2\pi s), 2s)^T$$

maps the interval $(0, 5)$ to a helix in $\mathbb{R}^3$. For a given choice of s, if we let the third coordinate vary from $2s$ to $2s + 1$ we would then trace out a ribbon in $\mathbb{R}^3$. To do this, let's add a second coordinate t which ranges from 0 to 1. We then have function $R(s, t) = (\cos(2\pi s), \sin(2\pi s), 2s + t)$ which maps the rectangle $(0, 5) \times (0, 2)$ to a curved ribbon shape in $\mathbb{R}^3$. To generate data on the ribbon, we draw **iid** points uniformly from $(0, 5) \times (0, 1)$ then apply the ribbon function. The results for $N = 1000$ samples are shown below.

```
N <- 1e4
s <- runif(N,0,5)
t <- runif(N,0,1)
ribbon <- cbind(cos(2*pi*s),sin(2*pi*s),2*s+t )
scatterplot3js(ribbon,
               xlab = expression(x[1]), ylab = expression(x[2]), zlab = expression(x[3]),
               angle = 90,
               pch = '.',
               size = 0.1)
```

In both the surface is two-dimensional (if you were stand on the surface and look nearby data points, they would appear to be planar. A dimension reduction would be able to recover this two-dimensional structure which in the case of the plane corresponds to find the coefficients used to generate each point. For the ribbons, the coordinates (s, t) used in the ribbon mapping would provide the simplified representation.

The previous examples have shown data in $\mathbb{R}^3$. While this is helpful of visualization, it is not representative of many modern data sources where the dimensionality is extremely high! Suppose we want to generate data in $\mathbb{R}^d$ for some specific large d which has a simpler representation in $\mathbb{R}^t$ (with $t \ll d$) that we know.

In general we can do this through a mapping, $f : \mathbb{R}^t \to \mathbb{R}^d$. (More on this idea in **??**ch-nonlinear) later). First generate synthetic data $\vec{x}'_1, \ldots, \vec{x}'_N$ then apply

Figure 3.2: Realizations of points on a ribbon

the map to create high dimensional observations

$$\vec{x}_i = f(\vec{x}'_i), \qquad i = 1, \ldots, N.$$

There are infinite possibilities for f, but for now, we will focus on affine mappings. For a matrix $\mathbf{A} \in \mathbb{R}^{d \times t}$ and vector $\vec{b} \in \mathbb{R}^d$, the affine function

$$f(\vec{x}') = \mathbf{A}\vec{x}' + \vec{b}$$

maps vectors in $\mathbb{R}^t$ to points in $\mathbb{R}^d$. Properties of this mapping with depend on the columns of $\mathbf{A}$. For example, if the columns are orthonormal, then Euclidean distances between corresponding pairs of vectors will be the same in the low and high dimensional spaces.

Example 3.3 (Mapping unit sphere in $\mathbb{R}^3$ to higher dimensions)**.** We begin be drawing sample from $\mathcal{N}(\vec{0}, \mathbf{I}_3)$ then rescaling each vector to have length 1.

```
N <- 500
X <- mvrnorm(n=N, mu = rep(0,3), Sigma = diag(1,3))
for (i in 1:N){
  X[i,] <- X[i,]/sqrt(sum(X[i,]^2))
}
scatterplot3js(X,
```

```
              xlab = expression(x[1]), ylab = expression(x[2]), zlab = expression(x[3]),
              angle = 90,
              pch = '.',
              size = 0.1)
```

Below, we apply the affine mapping when the columns of $\mathbf{A}$ are orthonormal and $\vec{b} = \vec{0}$ and show a few samples.

```
U <- svd(matrix(rnorm(10*10),nrow = 10))$u[,1:3]
X <- X %*% t(U)
head(round(X,3))
```

```
##        [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]   [,9]  [,10]
## [1,]  0.279  0.678 -0.181  0.252 -0.225  0.173 -0.143 -0.348 -0.284  0.252
## [2,] -0.255 -0.686  0.100 -0.236  0.197 -0.291  0.148  0.318  0.260 -0.292
## [3,]  0.084  0.771  0.043  0.216 -0.415  0.002 -0.124  0.239 -0.243  0.215
## [4,]  0.163 -0.389 -0.500  0.007  0.110 -0.495  0.065 -0.488 -0.033 -0.269
## [5,]  0.254  0.637 -0.321  0.268 -0.365 -0.255 -0.111 -0.186 -0.317  0.097
## [6,]  0.223  0.618 -0.319  0.260 -0.399 -0.347 -0.099 -0.094 -0.310  0.060
```

In the above example, we generated a random orthonormal matrix by applying the SVD to a random 10×10 matrix with $\mathcal{N}(0, 1)$ entries. I could continue applying additional tranformations (shifts and expansions/contractions along different directions), but we will stop for now with the following important observations. Here is a dataset which has structure we know, 2-dimensional data on a curved surface (the sphere), but we cannot determine this but looking at

the data matrix or generating scatterplots of any subset of columns. An effective way of generating data with known but hidden structure which is great to testing algorithms (and students).

3.3.2 Clustered data

The most straightforward way to generate clustered data is to combine realizations from separate data generating mechanisms that tend to create points in disjoint regions of $\mathbb{R}^d$. In Figure 3.3, we show two different cases in $\mathbb{R}^2$.

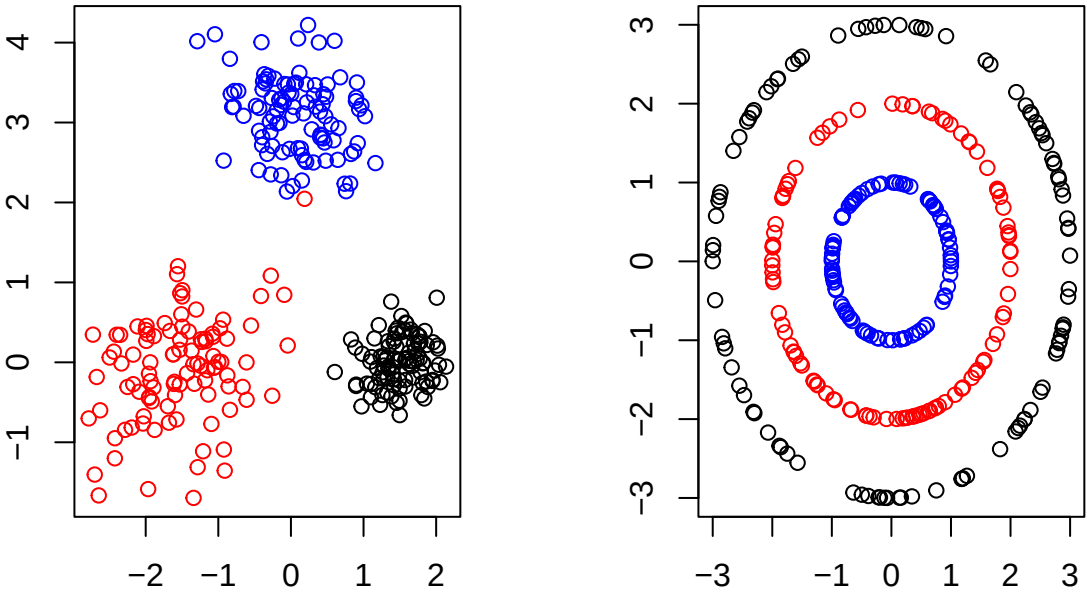


Figure 3.3: In each subfigure below, different subsets of points were generated using different rules and are colored accordingly . Ideally, a clustering algorithm could detect the different cluster shapes (left: ellipsoids, right: concentric rings) and correctly group points depending on how they were generated.

If one did not have access to the actual data generating process (depicted by the different colors), it is still likely that they could recover the correct groupings upon visual inspection. In general, this strategy is not tractable. Naturally, we would like an Unsupervised clustering algorithm that can learn these clusters directly from the data automatically. As we shall see in Chapter 7, certain algorithms will excel at grouping the data contained in disjoint ellipsoids will naturally struggle data clustering in concentric rings because the shape(s) of the different clusters matters has a major impact of the accuracy of the clustering.

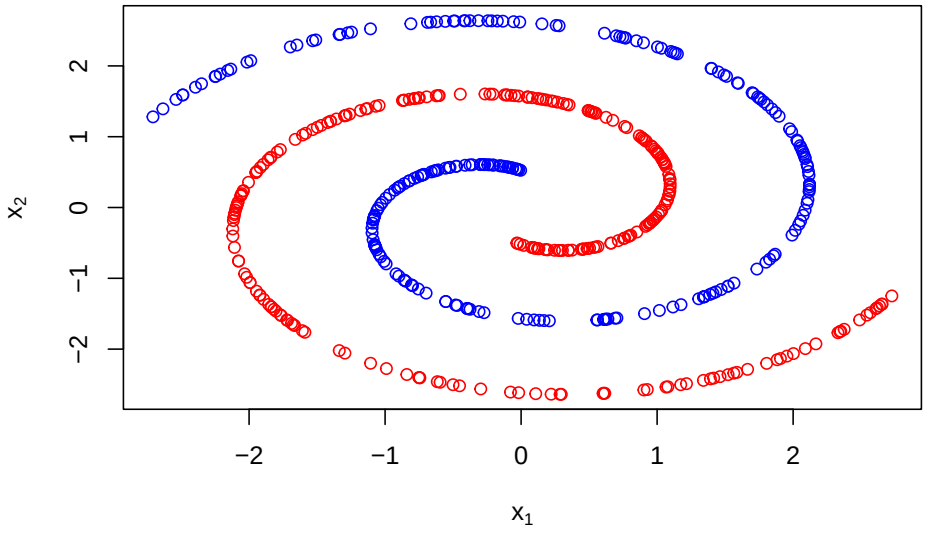
In both examples above a two step sampling procedure was used to generate the observations in different clusters. First, a latent cluster *label* was generated.

Then based on that label, an associated data point was created. Below we show an example which generates data in one of two spirals. For sample i, we first draw $z_i \sim Ber(1/2)$. Then if $z_i = 0$, we randomly generate x_i in the first spiral, otherwise we generate x_i in the second.

Example 3.4 (Intertwined spirals)**.**

```
# define functions for different spirals
spiral1 <- function(u){c(u*cos(3*u),u*sin(3*u))}
spiral2 <- function(u){c(u*cos(3*(u+pi)),u*sin(3*(u+pi)))}
# sample latent cluster assignments
z <- sample(c(0,1),size = 1e3,prob = c(1/2,1/2),replace = T)

# generate spirals based on labels
X <- matrix(NA,nrow = N,ncol = 2)
for (i in 1:N){
  if (z[i]==0)
    X[i,] <- spiral1(runif(1,0.5,3))
  else
    X[i,] <- spiral2(runif(1,0.5,3))
}
#plot
plot(X, col = c("blue","red")[z+1], xlab =expression(x[1]),ylab =expression(x[2]))
```



3.4 Exercises

1. Generate data on a helix in $\mathbb{R}^3$.
2. Generate data on a 7 dimensional hyperplane in $\mathbb{R}^{50}$. No columns of the

associated data matrix should contain all zeros.

3. Generate 2 well separated spherical clusters in $\mathbb{R}^3$. What is the Euclidean distance between the two closest points in different clusters?

4. Repeat problem 4 for $\mathbb{R}^{10}$ and $\mathbb{R}^{100}$. What changes did you have to make to keep the clusters well separated (at least as far apart as problem 3)?

Chapter 4

Linear Methods

4.1 Principal Component Analysis

Principal component analysis (PCA) is arguably the first method for dimension reduction, which dates back to papers by some of the earliest contributors to statistical theory including Karl Pearson and Harold Hotelling [**??**]. Pearson's original development of PCA borrowed ideas from mechanics which provides a clear geometric/physical interpretation of the resulting PCA **loadings**, **variances**, and **scores**, which we will define later. This interpretability and an implementation that uses scalable linear algebra methods – allowing PCA to be conducted on massive datasets – is one of the reasons PCA is still used prolifically to this day. In fact, many more modern and complex methods still rely on PCA as an internal step in their algorithmic structure.

There are number of different but equivalent derivations of PCA including the minimization of least squares error, covariance decompositions, and low rank approximations. We'll revisit these ideas later, but first, let's discuss PCA through a geometrically motivated lens via a method called iterative projections.

4.1.1 Derivation using Iterative Projections

We begin with a data matrix

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1^T \\ \vdots \\ \vec{x}_N^T \end{bmatrix} \in \mathbb{R}^{N \times d}.$$

Let's begin with an example of dimension reduction where we'll seek to replace each vector $\vec{x}_1, \ldots, \vec{x}_N$ with corresponding scalars $y_1, \ldots, y_N$ which preserve as much of the variability between these vectors as possible. To formalize this idea, let's introduce a few assumptions.

First, we'll assume the data $\vec{x}_1, \ldots, \vec{x}_N$ are centered. To be fair, assumption may be too strong a word. Rather, it's a first step in preprocessing that will simplify the analysis later. We'll discuss how to account for this centering step later, but for now assume $\bar{x} = \vec{0}$ so that $\mathbf{HX} = \mathbf{X}$. More importantly, let's assume that each y_i is constructed in the same way. Specifically, let $y_i = \vec{x}_i^T \vec{w}$ for some common vector $\vec{w}$. Thus, we can view each one-dimensional representation as a dot product of the corresponding observed vector with the same vector $\vec{w}$. We can compactly write this expression as

$$\vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \vec{x}_1^T \vec{w} \\ \vdots \\ \vec{x}_N^T \vec{w} \end{bmatrix} = \mathbf{X}\vec{w}.$$

How do we choose $\vec{w}$? We would like differences in the scalars $y_1, \ldots, y_N$ to reflect differences in the vectors $\vec{x}_1, \ldots, \vec{x}_N$ so having $y_1, \ldots, y_N$ spread out is a natural goal. Thus, if $\vec{x}_i$ and $\vec{x}_j$ are far apart then so will y_i and y_j. To do this, we'll try to maximize the sample variance of the y's. The sample variance

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2 = \frac{1}{N}\sum_{i=1}^{N}(\vec{x}_i^T\vec{w} - \bar{y})^2$$

will depend on our choice of $\vec{w}$. In the previous expression,

$$\bar{y} = \frac{1}{N}y_i = \frac{1}{N}\sum_{i=1}^{N}\vec{x}_i^T\vec{w} = \frac{1}{N}\vec{1}^T\mathbf{X}\vec{w}$$

is the sample mean of $y_1, \ldots, y_N$. Importantly, since we have assumed that $\vec{x}_1, \ldots, \vec{x}_N$ are centered, it follows that $\bar{y} = 0$ and the sample variance of $y_1, \ldots, y_N$ simplifies to

$$\frac{1}{N}\sum_{i=1}^{N}(\vec{x}_i^T\vec{w})^2 = \frac{1}{N}\sum_{i=1}^{N}y_i^2 = \frac{1}{N}\|y\|^2 = \frac{1}{N}\vec{y}^T\vec{y}.$$

We can write the above in terms of $\mathbf{X}$ and $\vec{w}$. Using the identity $\vec{y} = \mathbf{X}\vec{w}$, we want to choose $\vec{w}$ to maximize

$$\frac{1}{N}\vec{y}^T\vec{y} = \frac{1}{N}(\mathbf{X}\vec{w})^T\mathbf{X}\vec{w} = \frac{1}{N}\vec{w}^T\mathbf{X}^T\mathbf{X}\vec{w} = \vec{w}^T\left(\frac{\mathbf{X}^T\mathbf{X}}{N}\right)\vec{w}.$$

Since we have assumed that $\mathbf{X}$ is centered it follows that $\mathbf{X}^T\mathbf{X}/N$ is the sample covariance matrix $\hat{\mathbf{\Sigma}}$! Thus, we want to make $\vec{w}^T\hat{\mathbf{\Sigma}}\vec{w}$ as large as possible.

Naturally, we could increase the entries in $\vec{w}$ and increase the above expression without bound. To make the maximization problem well posed, we will restrict $\vec{w}$ to be unit-length under the Euclidean norm so that $\|\vec{w}\| = 1$. We now have a constrained optimization problem which gives rise to the first **principal component loading**.

Definition 4.1 (First PCA Loading and Scores)**.** The first **principal component loading** is the vector $\vec{w}_1$ solving the constrained optimization problem

$$\begin{aligned} &\text{Maximize } \vec{w}^T \hat{\mathbf{\Sigma}} \vec{w} \\ &\text{subject to constraint } \|\vec{w}\| = 1. \end{aligned} \tag{4.1}$$

The first **principal component scores** are the projections, $y_i = \vec{x}_i^T \vec{w}_1$ for $i = 1, \ldots, N$, of each sample onto the first loading.

To find the first PCA loading we can make use of Lagrange multipliers (see exercises) to show that $\vec{w}_1$ must also satisfy the equation

$$\hat{\mathbf{\Sigma}} \vec{w}_1 = \lambda \vec{w}_1$$

where λ is the Lagrange multiplier. From this expression, we can conclude that the first principal component loading is the unit length eigenvector associated with the largest eigenvalue of the sample covariance matrix $\hat{\mathbf{\Sigma}}$ and that the Lagrange multiplier λ is the largest eigenvalue of $\hat{\mathbf{\Sigma}}$. In this case, we refer to λ as the first **principal component variance**.

4.1.1.1 Geometric Interpretation of $\vec{w}_1$

Since $\|\vec{w}_1\| = 1$ we may interpret this vector as specifying a direction in $\mathbb{R}^d$. Additionally, we can decompose each of our samples into two pieces: one pointing in the direction specified by $\vec{w}_1$ and a second portion perpendicular to this direction. Thus, we may write

$$\vec{x}_i = \underbrace{\vec{w}_1 \vec{x}_i^T \vec{w}_1}_{parallel} + \underbrace{(\vec{x}_i - \vec{w}_1 \vec{x}_i^T \vec{w}_1)}_{perpendicular}.$$

By the Pythagorean theorem,

$$\begin{aligned} \|\vec{x}_i\|^2 &= \|\vec{w}_1 \vec{x}_i^T \vec{w}_1\|^2 + \|\vec{x}_i - \vec{w}_1 \vec{x}_i^T \vec{w}_1\|^2 \\ &= (\vec{w}_1^T \vec{x}_i)^2 + \|\vec{x}_i - \vec{w}_1 \vec{x}_i^T \vec{w}_1\|^2 \\ &= y_i^2 + \|\vec{x}_i - \vec{w}_1 \vec{x}_i^T \vec{w}_1\|^2 \end{aligned}$$

for $i = 1, \ldots, N$. Averaging over all of samples gives the expression

$$\frac{1}{N} \sum_{i=1}^{N} \|\vec{x}_i\|^2 = \frac{1}{N} \sum_{i=1}^{N} y_i^2 + \frac{1}{N} \sum_{i=1}^{N} \|\vec{x}_i - \vec{w}_1 \vec{x}_i^T \vec{w}_1\|^2.$$

The left-hand side of the above expression is fixed for a given set of data, whereas the first term on the right side is exactly what we sought to maximize when finding the first principal component loading. This quantity is the average squared length of the projection of each sample onto the direction $\vec{w}_1$. As such, we can view the first principal component loading as the direction in which $\vec{x}_1, \ldots, \vec{x}_N$ most greatly varies. Let's turn to an example in $\mathbb{R}^3$ to view this.

Example 4.1 (Computing the First PCA Loading and Scores)**.** Below, we show a scatterplot of $N = 500$ points in $\mathbb{R}^3$ drawn randomly from a MVN. These data have been centered.

Notice the oblong shape of the cloud of points. Rotating this image, it is clear that the data varies more in certain directions than in others.

The sample covariance matrix of these data is

$$\hat{\Sigma} = \begin{bmatrix} 11.4 & 7.12 & 3.22 \\ 7.12 & 9.21 & 9.03 \\ 3.22 & 9.03 & 15.99 \end{bmatrix}$$

We can use the eigendecomposition of this matrix to find the first PCA loading and variance. Given the first loading we can also compute the first PCA scores. A short snippet of code for this calculation is shown below.

```
Sigmahat <- (t(data) %*% data)/N # calculate the sample covariance matrix, recall the
Sigmahat.eigen <- eigen(Sigmahat) # calculate the eigen decomposition of Sigmahat
y <- data %*% Sigmahat.eigen$vectors[,1] # calculate the scores
```

The largest eigenvalue (first PC variance) is $\lambda = 25.58$ with associated eigenvector $(-0.44, -0.57, -0.69)^T$ which is the first PC loading.

We can visualize these results in a few different ways. First, we can add the span of $\vec{w}_1$ (shown in red) to the scatterplot of the data. One can see that $\vec{w}_1$ is oriented along the direction where the data is most spread out.

We could also generate a scatterplot of the scores, but we'll show these scores along the $\vec{w}_1$ axes so that they correspond to the projection of each sample onto span$\{\vec{w}_1\}$; equivalently, we are plotting the vectors $y_i\vec{w}_i$.

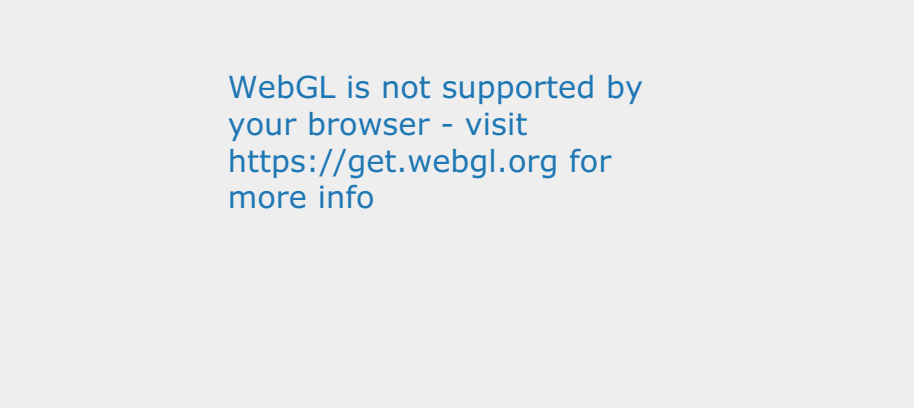


Figure 4.1: Decomposition of Samples into Components Parallel and Perpendicular to First Loading

4.1.1.2 Additional Principal Components

The first PCA loading provides information about the direction in which are data most greatly vary, but it is quite possible that there are still other directions

wherein our data still exhibits a lot of variability. In fact, the notion of a *first* principal component loading, scores, and variance suggests the existence of a second, third, etc. iteration of these quantities. To explore these quantities, let's proceed as follows

For each datum, we can remove its component in the direction of $\vec{w}_1$, and focus on the projection onto the orthogonal complement of $\vec{w}_1$. Let

$$\vec{x}_i^{(1)} = \vec{x}_i - \vec{w}_1 \vec{x}_i^T \vec{w}_1 = \vec{x}_i - \vec{w}_1 y_i$$

denote the portion of $\vec{x}_i$ which is orthogonal to $\vec{w}_1$. These points are shown on the right side of Fig @{fig:pca1}. Here, the superscript $^{(1)}$ indicates we have removed the portion of each vector in the direction of the first loading.

We can organize the orthogonal components into a new data matrix

$$\mathbf{X}^{(1)} = \begin{bmatrix} \left(\vec{x}_1^{(1)}\right)^T \\ \vdots \\ \left(\vec{x}_N^{(1)}\right)^T \end{bmatrix} = \begin{bmatrix} \vec{x}_1^T - \vec{x}_1^T \vec{w}_1 \vec{w}_1^T \\ \vdots \\ \vec{x}_N^T - \vec{x}_N^T \vec{w}_1 \vec{w}_1^T \end{bmatrix} = \mathbf{X} - \mathbf{X}\vec{w}_1 \vec{w}_1^T.$$

Now let's apply PCA to the updated data matrix $\mathbf{X}^{(1)}$ from which we get the second principal component loading, denoted $\vec{w}_2$, the second principal component scores, and the second principal component variance. One can show that the data matrix $\mathbf{X}^{(1)}$ is centered so that its sample covariance matrix is

$$\hat{\mathbf{\Sigma}}^{(1)} = \frac{1}{N}(\mathbf{X}^{(1)})^T \mathbf{X}^{(1)}. \tag{4.2}$$

The 2nd PC variance is the largest eigenvalue of $\hat{\mathbf{\Sigma}}^{(1)}$ and its associated unit length eigenvector is the 2nd PC loading, denoted $\vec{w}_2$. The second PC scores $\vec{w}_2^T \vec{x}_i^{(1)}$ for $i = 1, \ldots, N$.

Continuing **??**, we have

$$\hat{\mathbf{\Sigma}}^{(1)} = \begin{bmatrix} 6.36 & 0.61 & -4.61 \\ 0.61 & 0.81 & -1.07 \\ -4.61 & -1.07 & 3.86 \end{bmatrix}$$

which has largest eigenvalue 10.02 (2nd PC variance) and associated eigenvector $\vec{w}_2 = (0.78, 0.12, -0.61)^T$ (2nd PC Loading). The second set of PC scores are given by $\vec{w}_2^T \vec{x}_i^{(1)}$ for $i = 1, \ldots, N$.

Here is one crucial observation. The vector $\vec{w}_2$ gives the direction of greatest variability of the vectors $\vec{x}_1^{(1)}, \ldots, \vec{x}_N^{(1)}$. For each of these vectors we have removed the component in the direction of $\vec{w}_1$. Thus, $\vec{x}_1^{(1)}, \ldots, \vec{x}_N^{(1)}$ do not vary at all in the $\vec{w}_1$ direction. What can we say about $\vec{w}_2$? Naturally, it must be perpendicular to $\vec{w}_1$! We can see this geometric relationship if we plot the vectors $\vec{x}_i^{(1)}$ in our previous example along with the span of the first and second loading.

Figure 4.2: First and 2nd Loadings with Data after component along first loading has been removed

We need not stop at the second PCA loading, scores, and variance. We could remove components in the direction of $\vec{w}_2$ and apply PCA to the vectors

$$\begin{aligned}\vec{x}_i^{(2)} &= \vec{x}_i^{(1)} - \vec{w}_2(\vec{x}_i^{(1)})^T\vec{w}_2 \\ &= \vec{x}_i - \vec{w}_1\vec{x}_i^T\vec{w}_1 - \vec{w}_2(\vec{x}_i - \vec{w}_1\vec{x}_i^T\vec{w}_1)^T\vec{w}_2 \\ &= \vec{x}_i - \vec{w}_1\vec{x}_i^T\vec{w}_1 - \vec{w}_2\vec{x}_i^T\vec{w}_2 + \vec{w}_2\vec{w}_1{}^T\vec{x}_i\underbrace{\vec{w}_1^T\vec{w}_2}_{=0}\end{aligned}$$

which gives rise to the centered data matrix

$$\mathbf{X}^{(2)} = \begin{bmatrix}\vec{x}_1^T - \vec{w}_1^T\vec{x}_1\vec{w}_1^T - \vec{w}_2^T\vec{x}_1\vec{w}_2^T \\ \vdots \\ \vec{x}_d^T - \vec{w}_1^T\vec{x}_d\vec{w}_1^T - \vec{w}_2^T\vec{x}_d\vec{w}_2^T\end{bmatrix} = \mathbf{X} - \mathbf{X}\vec{w}_1\vec{w}_1^T - \mathbf{X}\vec{w}_2\vec{w}_2^T \tag{4.3}$$

with corresponding covariance matrix

$$\hat{\Sigma}^{(2)} = \frac{1}{N}(\mathbf{X}^{(2)})^T\mathbf{X}^{(2)}$$

from which we can obtain a third loading ($\vec{w}_3$), variance (λ_3), and set of scores $\vec{w}_3\vec{x}_i^{(2)}$ for $i = 1, \ldots, N$.

We can continue repeating this argument d times for our d-dimensional data until we arrive at a set of d unit vectors $\vec{w}_1, \ldots, \vec{w}_d$ which are the d PCA loadings. Why do we stop after d? The principal component loadings $\vec{w}_1, \ldots, \vec{w}_d$ are all mutually orthogonal and unit length so they form an orthonormal basis for $\mathbb{R}^d$. All of the variability in each sample can be expressed in terms of these d basis vectors.

This iterative approach is admittedly, to intensive for most practical applications. Fortunately, we do not need to follow the sequence of projections and then eigenvalue computations thanks to the following theorem.

Theorem 4.1. *Suppose $\vec{w}_1, \ldots, \vec{w}_d$ are the orthonormal eigenvectors of the sample covariance $\hat{\Sigma} = \frac{1}{N}\mathbf{X}^T\mathbf{X}$ with eigenvalues $\lambda_1 \geq \cdots \geq \lambda_d \geq 0$ respectively, i.e. $\hat{\Sigma}\vec{w}_j = \lambda_j \vec{w}_j$. Then $\vec{w}_1, \ldots, \vec{w}_d$ are also eigenvectors or the matrix $\hat{\Sigma}^{(1)} = \frac{1}{N}(\mathbf{X}^{(1)})^T\mathbf{X}^{(1)}$ with eigenvalues $0, \lambda_2, \ldots, \lambda_d$ respectively.*

This result follows nicely from the geometric observation that the loadings are mutually orthogonal. The second loading is then the eigenvector associated with the second largest eigenvalue of the original covariance matrix. Taking the orthogonality of the loadings further, we can get even more from the following corollary.

Corollary 4.1. *Suppose $\vec{w}_1, \ldots, \vec{w}_d$ are the orthonormal eigenvectors of the sample covariance $\hat{\Sigma} = \frac{1}{N}\mathbf{X}^T\mathbf{X}$ with eigenvalues $\lambda_1 \geq \cdots \geq \lambda_d \geq 0$ respectively. Then for $k = 2, \ldots, d-1$, the vectors$\vec{w}_1, \ldots, \vec{w}_d$ are also eigenvectors of the matrix $\hat{\Sigma}^{(k)} = \frac{1}{N}(\mathbf{X}^{(k)})^T\mathbf{X}^{(k)}$ with eigenvalues $\underbrace{0, \ldots, 0}_{k}, \lambda_{k+1}, \ldots, \lambda_d$ respectively.*

As a result, we can immediately compute the PCA variances and loadings given the full spectral (eigenvalue) decomposition

$$\hat{\Sigma} = \begin{bmatrix} \vec{w}_1 & | & \ldots & | & \vec{w}_d \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_d \end{bmatrix} \begin{bmatrix} \vec{w}_1^T \\ \vdots \\ \vec{w}_d^T \end{bmatrix}. \tag{4.4}$$

Again, thanks to orthogonality, we can also compute the PCA scores directly from the original data without iteratively removing components of each vector in the direction of the various loadings. To summarize PCA, given data $\mathbf{X}$ we first compute the spectral decomposition of the sample covariance matrix $\hat{\mathbf{\Sigma}}$. The eigenvalues (in decreasing magnitude) provide the PC variance and the corresponding unit-length eigenvectors give the corresponding loadings, which form an orthonormal basis in $\mathbb{R}^d$. The PC scores are the inner product of each vector with these loadings (assuming the data are centered). Thus, the PCA scores are $\vec{x}_i^T\vec{w}_1, \ldots, \vec{x}_i^T\vec{w}_d$ for $i = 1, \ldots, d$. We can compactly (again assuming our data are centered) express the scores in terms of the original data matrix and the loadings as

$$\mathbf{Y} = \mathbf{XW} \tag{4.5}$$

where $\mathbf{Y} \in \mathbb{R}^{N \times d}$ is a matrix of PC scores and $\mathbf{W}$ is the orthonormal matrix with columns given by the loadings.

4.1.1.3 Geometric interpretation of PCA

We began with an $N \times d$ data matrix $\mathbf{X}$ and now we have an $N \times d$ matrix of PCA scores $\mathbf{Y}$. One may be inclined to ask: where is the dimension reduction? To answer this question, let's revisit some important features of the scores and loadings.

First, since the loadings $\vec{w}_1, \ldots, \vec{w}_d$ are orthonormal vectors in $\mathbb{R}^d$, they naturally form a basis! The PCA scores are then the coordinates of our data in the basis $\{\vec{w}_1, \ldots, \vec{w}_d\}$. Specifically, $\mathbf{Y}_{ij}$ for $j = 1, \ldots, d$ are the coordinates for (centered) vector $\vec{x}_i$ since

$$\vec{x}_i = \mathbf{Y}_{i1}\vec{w}_1 + \cdots + \mathbf{Y}_{id}\vec{w}_d.$$

The dimension reduction comes in from an important observations about the PCA scores (aka coordinate). Consider the sample covariance matrix of the PCA scores, which one can show is diagonal (see exercises)

$$\hat{\mathbf{\Sigma}}_Y = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_d \end{bmatrix} \tag{4.6}$$

There are two important perspectives on the using PCA scores. First, the diagonality of the PCA scores indicates that in the new basis, $\{\vec{w}_1, \ldots, \vec{w}_d\}$, the coordinates are i) uncorrelated and ii) ordered in decreasing variance. As a result, if we want to construct a $k < d$ dimensional representation our data that minimizes the loss in variability, we should use the first k PCA scores.

Secondly, we can use the first k scores to build approximations,

$$\vec{x}_i \approx \hat{x}_i = \mathbf{Y}_{i1}\vec{w}_1 + \cdots + \mathbf{Y}_{ik}\vec{w}_k, \qquad i = 1, \ldots, N.$$

How good is this approximation? Optimal, in a squared loss sense, thanks to the following theorem.

Theorem 4.2. *Fix $k < d$. Given any k-dimensional linear subspace of $\mathbb{R}^d$, denoted $\mathcal{V}$, let $proj_{\mathcal{V}}(\vec{x})$ denote the orthogonal projection of $\vec{x}$ onto $\mathcal{V}$, then*

$$\frac{1}{N}\sum_{i=1}^{N} \|\vec{x}_i - proj_{\mathcal{V}}(\vec{x}_i)\|^2 \geq \lambda_{k+1} + \cdots + \lambda_d$$

where $\lambda_{k+1}, \cdots, \lambda_d$ are the last $(d-k)$ principal component variances of $\vec{x}_1, \ldots, \vec{x}_N$.

Furthermore, if $\lambda_k > \lambda_{k+1}$, then span$\{\vec{w}_1, \ldots, \vec{w}_k\}$ is the unique k-dimensional linear subspace for which

$$\frac{1}{N}\sum_{i=1}^{N} \|\vec{x}_i - proj_{\mathcal{V}}(\vec{x}_i)\|^2 = \lambda_{k+1} + \cdots + \lambda_d.$$

A few notes on the above theorem to add some clarity. First, note that

$$\hat{x}_i = \mathbf{Y}_{i1}\vec{w}_1 + \cdots + \mathbf{Y}_{ik}\vec{w}_k$$

is the orthogonal projection of $\vec{x}_i$ onto $\text{span}\{\vec{w}_1, \ldots, \vec{w}_k\}$ so that

$$\vec{x}_i - \text{proj}_{\text{span}\{\vec{w}_1,\ldots,\vec{w}_k\}}(\vec{x}_i) = \mathbf{Y}_{i,k+1}\vec{w}_{k+1} + \cdots + \mathbf{Y}_{id}\vec{w}_d.$$

Using this identity and the properties of the PCA scores allows one to verify the equality at the end of the theorem (see exercises).

The requirement that $\lambda_k \geq \lambda_{k+1}$ is necessary for uniqueness. If $\lambda_k = \lambda_{k+1}$ then the subspace $\text{span}\{\vec{w}_1, \ldots, \vec{w}_{k-1}, \vec{w}_{k+1}\}$ would offer an equally good approximation. Fortunately, it is rare that two (nonzero) PCA scores are equal in practice so we rarely have to worry about this issue. In short, PCA learns the k-dimensional linear subspace which best approximates the original data on average!

In all of the previous analysis, we have been operating under the assumption that our data is centered, e.g. $\bar{x} = \vec{0}$ – an preprocessign step that made our analysis easier and is done in practice to avoid potential issue with numerical linear albegra. We can recover all of the nice geometric intuition when working with uncentered data by adding the sample mean back to the approximation and shifting our focus to affine subspaces, which look just like linear subspaces which have been translated by some constant shift away from the origin.

::: {.corollary $pca-opt-aff} Given any k-dimensional affine subspace $\mathcal{A} \subset \mathbb{R}^d$

$$\frac{1}{N}\sum_{i=1}^{N} \|\vec{x}_i - \text{proj}_{\mathcal{A}}(\vec{x}_i)\|^2 \geq \lambda_{k+1} + \cdots + \lambda_d$$

with equality (assuming $\lambda_k > \lambda_{k+1}$) if and only if $\mathcal{A} = \bar{x} + \text{span}\{\vec{w}_1, \ldots, \vec{w}_k\}$. :::

Note the set $\bar{x} + \text{span}\{\vec{w}_1, \ldots, \vec{w}_k\}$ is the set of all vectors

$$\{\vec{v} \in \mathbb{R}^d : \vec{v} - \bar{x} \in \text{span}\{\vec{w}_1, \ldots, \vec{w}_k\}\},$$

i.e. the optimal k-dimensional hyperplane found after centering if translated by $\hat{x}$. In this case, the optimal approximation of our data is

$$\vec{x}_i \approx \bar{x} + \mathbf{Y}_{i1}\vec{w}_1 + \cdots + \mathbf{Y}_{ik}\vec{w}_k.$$

4.1.2 PCA in Practice

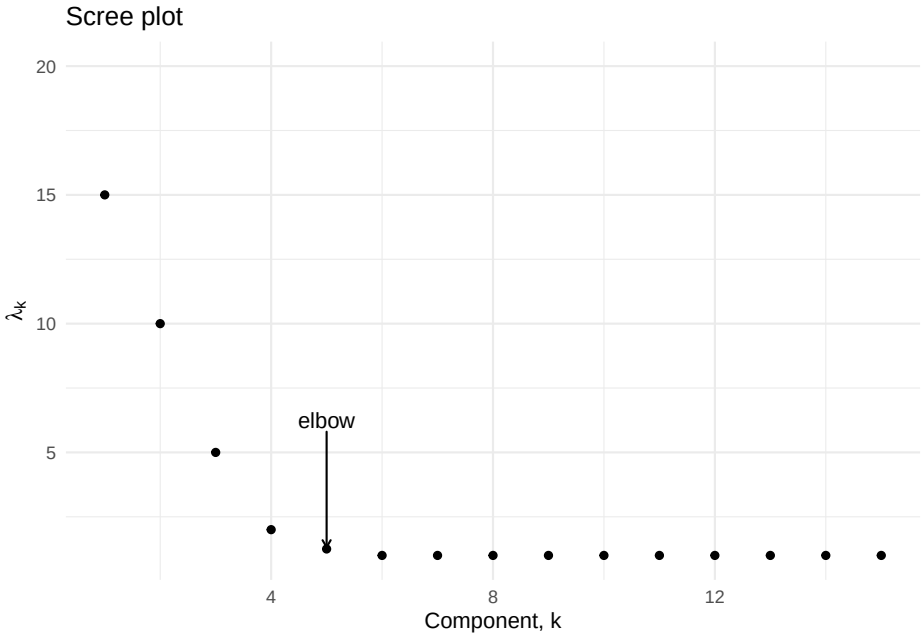
4.1.2.1 Choosing the number of components

Now that we have explored the mathematical details of PCA, let us turn to practical guidance when working with real data.

A natural first question for PCA, is how many principal components should I use? Like many modeling decisions in statistics there are a range of answers from simple heuristics to advanced technical answers. For PCA, methods based on random matrix theory are quite strong but also beyond the scope of this book. The curious reader is encouraged to see [**?**] for more information. Herein, we are going to focus on simpler but more common approaches with an emphasis on inherent trade-offs. Like many statistical modeling choices, decisions are not always so clear cut, and it's best to understand and reason about decisions in the context of a specific problem rather than following hard and fast rules.

The best known method for choosing an optimal number of principal components is the scree plot [**?**], which is based on a visual inspection of a plot of the principal component variances in decreasing order. Ideally, one would observe an *elbow* delineating a regions where the variances decrease sharply then saturate near a small value.

```
library("ggrepel")
df <- data.frame(lambda = c(15,10,5,2,1.25,1,1,1,1,1,1,1,1,1,1), comp = 1:15)
elbow_point <- 5
ggplot(df, aes(x=comp, y = lambda)) + geom_point() + xlab("Component, k") +
  ylab(expression(lambda[k])) +
  ggtitle("Scree plot") +
  theme_minimal() +
  # Annotate the elbow point
  geom_text_repel(aes(label = ifelse(comp == elbow_point, "elbow", "")),
                  nudge_y = 5,
                  arrow = arrow(length = unit(0.015, "npc")))
```



In the preceding figure, we would opt to keep the first four components since the variance appears to saturate beginning with the fifth principal component.

Another heuristic suggests keeping the number of components needed to explain a specified percentage of the variance, typically 80 or 90%. Using 80% as a guide would suggest keeping the first 7 principal components (using the first 6 only gets us to 79.2%) based on the related figure below.
Using the same data as above, we have the following *percentage explained variance* plot below

```
df$varp = cumsum(df$lambda)/sum(df$lambda)
ggplot(df, aes(x=comp, y = varp)) + geom_point() + xlab("Component, k") +
  ylab("% of explained variance")
```

Specifically, we have plotted the cumulative sum of the p.c. variances relative to the total variance

$$\frac{\sum_{j=1}^{k}\lambda_j}{\sum_{j=1}^{d}\lambda_j} = \frac{\sum_{j=1}^{k}\lambda_j}{tr(\hat{\mathbf{\Sigma}}_x)}$$

as a function of the number of components.

Based on an evaluation of the principal component variances we have two heuristics suggesting different cutoffs (4 vs. 7). Which (if either) is correct? Like many issues in statistics, the answer depends on context. Perhaps there is a data driven reason to choose 4 or 7 or some other number for that matter. The important issue is to recognize the inherent trade-off in the choice so that one can make the best decision in the context of the larger data science problem where PCA is being applied. Choosing a smaller (larger) number of components

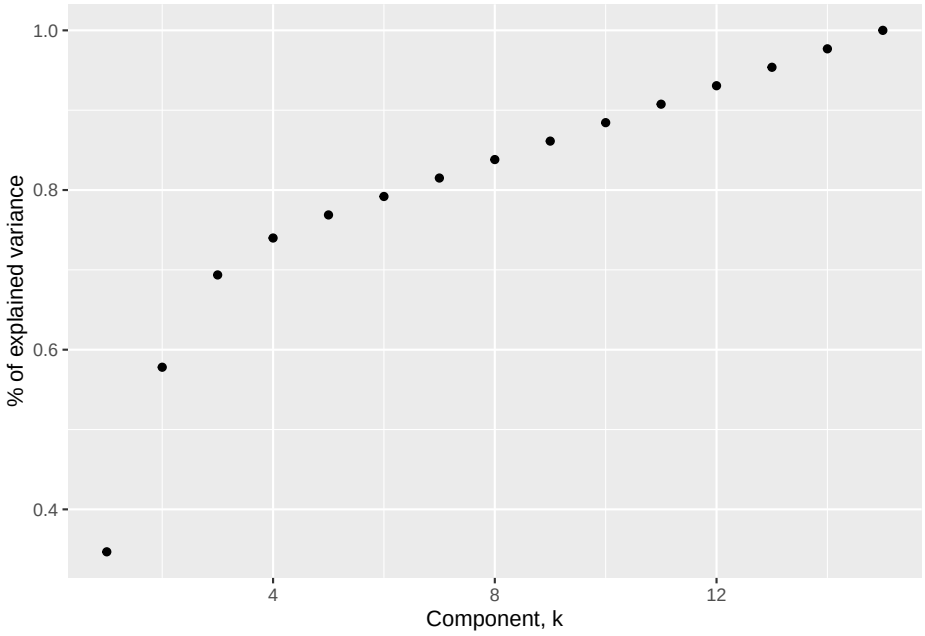


Figure 4.3: Percentage of explained variance

gives a greater level of dimension reduction at the expense of worse (better) approximation and retention of variability.

Additionally, the variances alone often are not enough to understand the structure of your data. As we will see in the next section, in some cases you may observe as stark drop in variances that is the result of features of data unrelated to concentration near a lower dimensional hyperplane.

4.1.2.2 Limitations and vulnerabilities

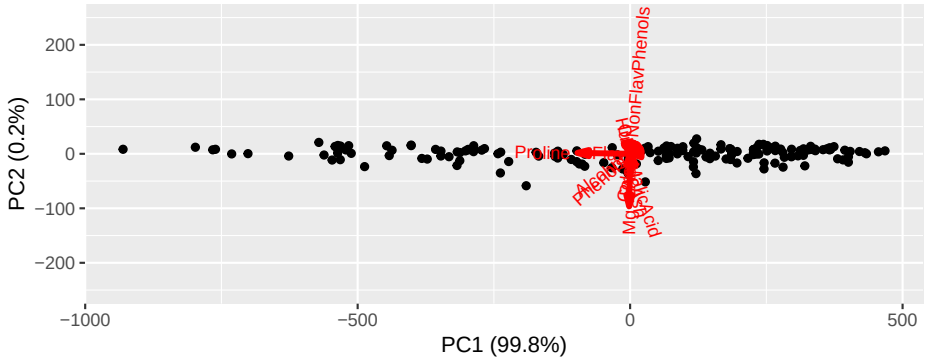
PCA relies on sample variance of data projected in different direction. As such, it is susceptible to outliers and/or imbalanced scales which can greatly influence calculations of sample variance. Furthermore, PCA is optimal at finding the best *linear* subspace as shown in the preceding sections. However, it cannot find (potentially simpler and more compressed) nonlinear structure in the data. Let's investigate each of these limitations in the following subsections.

4.1.2.2.1 Imbalanced scales As a motivating example, let's consider the wine dataset which contains observations of 13 chemical compounds (variables) for 178 different wines (subjects) from three different cultivars. Below we show the first 10 rows of the data matrix.

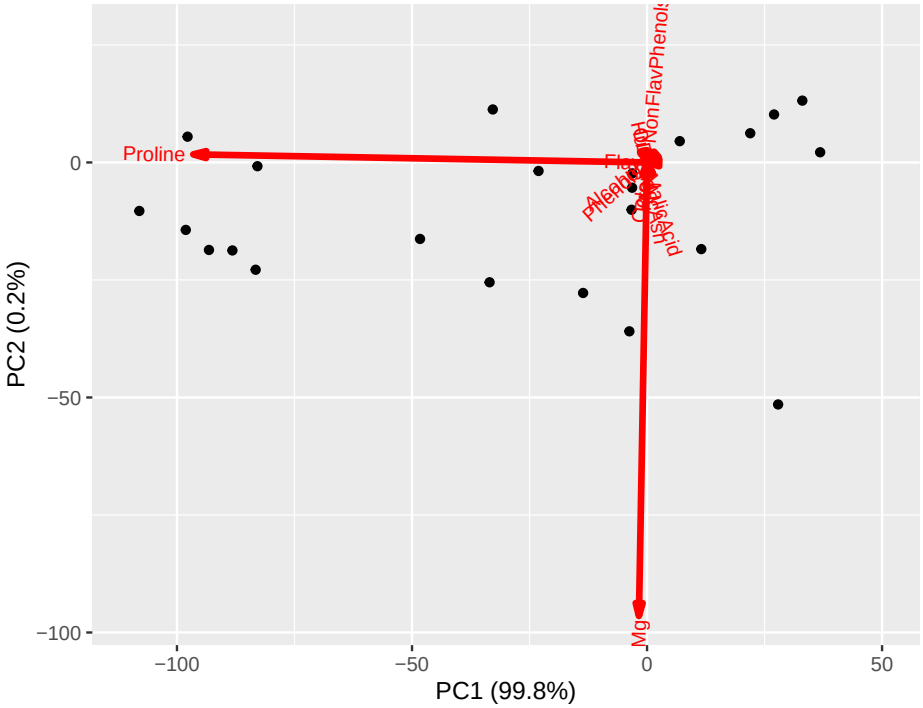
Alcohol	MalicAcid	Ash	AlcAsh	Mg	Phenols	Flav	NonFlavPhenols	Proa	Color	Hu
14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.0
13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.0
13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.0
14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.8
13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.0
14.20	1.76	2.45	15.2	112	3.27	3.39	0.34	1.97	6.75	1.0
14.39	1.87	2.45	14.6	96	2.50	2.52	0.30	1.98	5.25	1.0
14.06	2.15	2.61	17.6	121	2.60	2.51	0.31	1.25	5.05	1.0
14.83	1.64	2.17	14.0	97	2.80	2.98	0.29	1.98	5.20	1.0
13.86	1.35	2.27	16.0	98	2.98	3.15	0.22	1.85	7.22	1.0

Prior to running PCA, we might be able to guess which coordinates account for the most variability. Proline (on the scale of 1000s) and magnesium (on the scale of 100s) have much larger values than other coordinates in the data matrix. We can see this observation holds out immediately when looking at the *biplot* summarizing PCA run on the wine data.

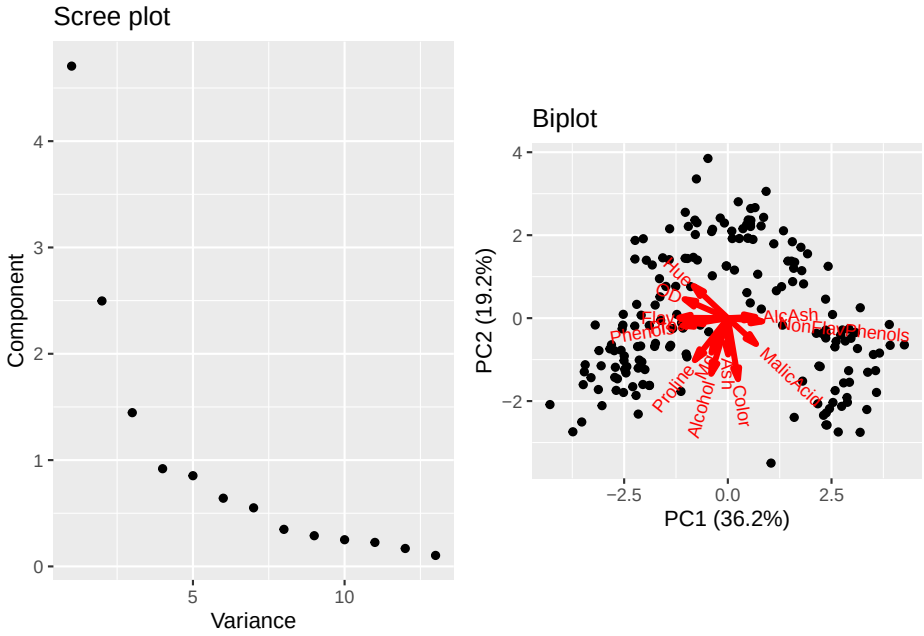
```
library(ggbiplot)
out <- prcomp(wine)
ggbiplot(out, scale = 0, varname.color = "red")  +ylim(c(-250,250))
```



In the figure above, the black dots are scatterplots of the first and second principal component score for each sample in our data set, i.e. $\{(y_{i1}, y_{i2})\}_{i=1}^{N}$, whereas the arrows correspond to the contribution of each column of the data matrix to the first and second loadings, i.e. $\{(\vec{w}_1)_j, (\vec{w}_2)_j\}_{j=1}^{d}$. The red vector associated with proline is the only one with a large horizontal component, indicating it is essentially the only coordinate contributing to the first loading. A similar observation indicates magnesium is the only coordinate contributing to the second loading. Admittedly, the arrow corresponding to the variables are difficult to see, but we can zoom in closer to the origin where the results are clear.



A natural way (and generally good idea) to address this issue is to standardize the columns of the data matrix prior to running PCA. If we use this approach for the wine data, we get the following biplot where there is much more diverse contributions to the first and second loadings from many coordinates.



Outliers

The existence of an outlier in a dataset (as in the univariate case) will alter the calculation of a sample covariance. However, a point can be an outlier because it has an extreme value in one **or more** of its coordinates. Unfortunately, standardization will not do much to alter this issue. Nonethess, the biplot is a helpful method for identifying this issue. Consider the wine data set, but let us now append a new data vector. It has the same values as the first row, except let's set the malic acid and ash columns to 10^6 (these are unrealistic extreme values only intended to emphasis the effect of an outlier). Below we show the scree plot and biplot resulting from PCA without standardization.

Here, we see one point has an extreme PC score associated with the outlier, which also accounts for a single large variance. It would be inappropriate to assume that a single PC component is sufficient to describe the data. Additionally, observe that the first loading is dominated by Malic Acid and Ash, values for which the outlier is extreme. Standardizing the coordinates can reduce the impact of the outlier but cannot remove it entirely. In such as case, further investigation to see the specific impact the outlier has on the loadings (and the potential overemphasis on some features) warrants further investigation.

4.1.2.2.2 Nonlinear structure Approximations using PCA are built via linear combinations of the loadings. However, there is not reason to *a priori* expect that the lower dimensional structure of our data is inherently linear. As such, PCA (and the other linear methods we'll discuss in the remainder of this chapter) are limited in the complexity of structure they can discover in data. As a final example, consider the following data on a curve in $\mathbb{R}^3$.

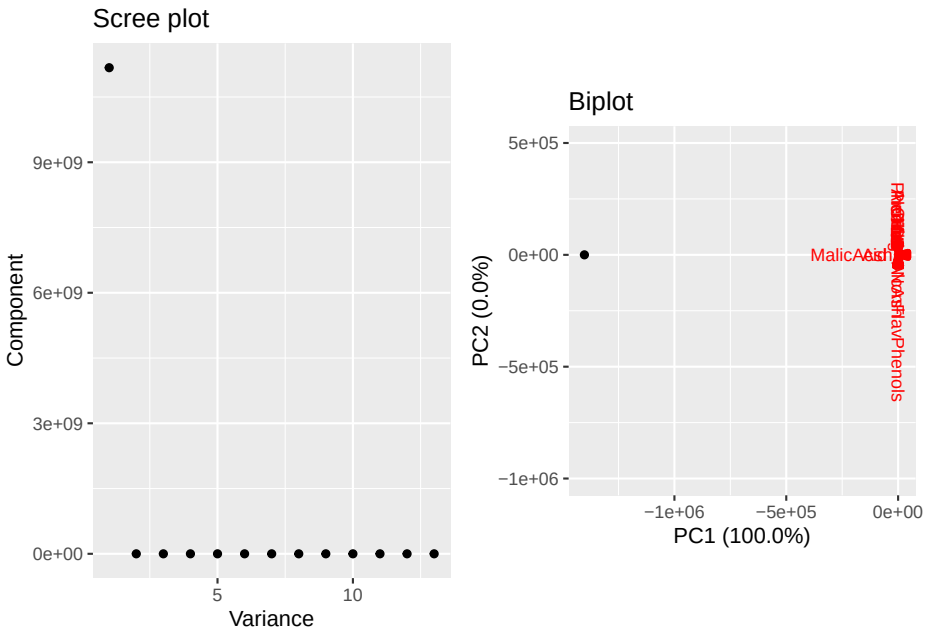


Figure 4.4: PCA on wine data with an outlier

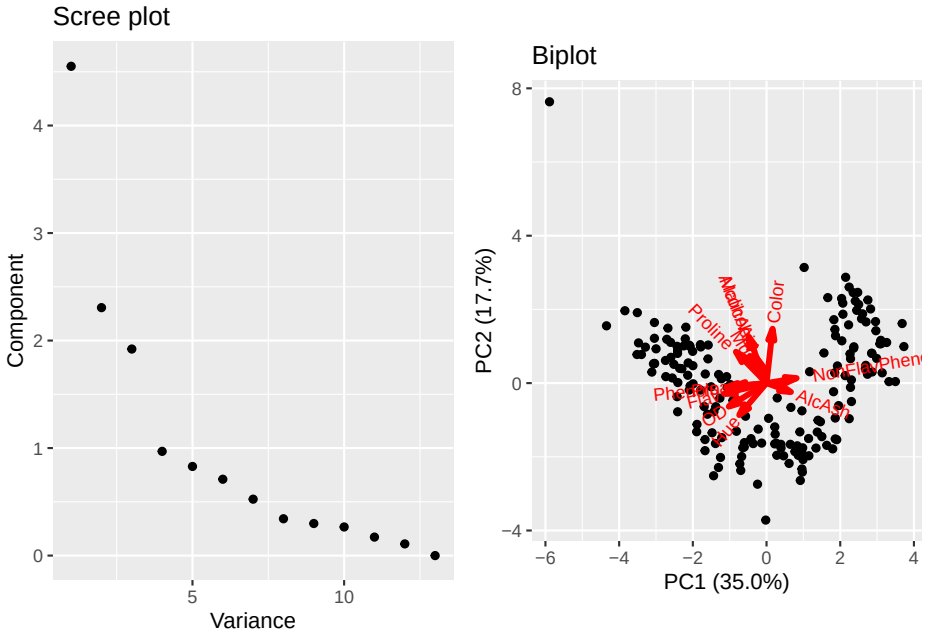
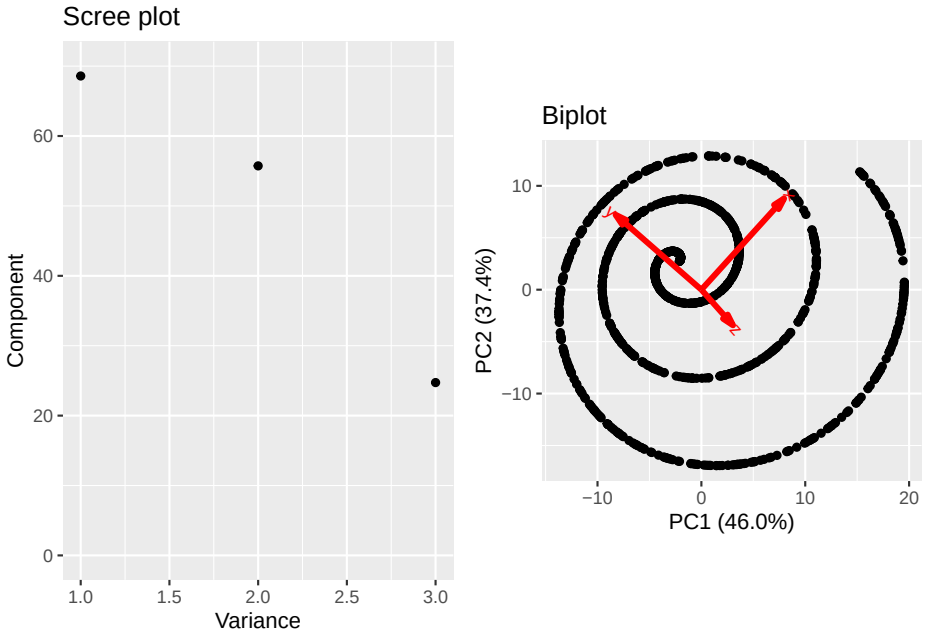


Figure 4.5: PCA on wine data with outlier after standardization

These data reside on a one-dimensional curve, specifically the curve parameterized by the equation

$$t \mapsto (t\cos(t), t\sin(t), t^2/20)^T.$$

However, if we inspect the principal component variance, we do not see a clear separation in the first PC variance with the second and third. There is a jump between the second and third, and plotting the first two scores (biplot) shows the nonlinear structure.



Ultimately, the fundamental limitation of PCA is a direct byproduct of it's primary strength. PCA discovers best approximating hyperplanes but nothing more complex!

4.2 Singular Value Decomposition

4.2.1 Low-rank approximations

In the next two subsections, we are going to focus on *low-rank* matrix approximation methods in which we try to approximate our data matrix $\mathbf{X}$ using a low-rank alternative. In the language of dimension reduction, the idea is to approximate each data with a linear combination of a small number ($k < d$ of latent feature vectors. Briefly, let's discuss how this idea works in the case of PCA.

In PCA, the loadings provide a data-driven orthonormal basis $\vec{w}_1, \ldots, \vec{w}_d$ which allow us to compute the PCA scores from the centered data. In matrix notation, this scores are given by

$$\underbrace{\mathbf{Y}}_{\text{PCA scores}} = \underbrace{(\mathbf{HX})}_{\text{centered data matrix}} \times \underbrace{\mathbf{W}}_{\text{loadings}}.$$

The matrix $\mathbf{W}$ is orthonormal allowing us to write

$$\mathbf{HX} = \mathbf{YW}^T.$$

The *ith* row of the preceding matrix equality reads

$$(\vec{x}_i - \bar{x})^T = \sum_{j=1}^{d} \mathbf{Y}_{ij} \vec{w}_j^T.$$

From the PCA notes, an approximation using the first k loadings

$$(\vec{x}_i - \bar{x})^T \approx \sum_{j=1}^{k} \mathbf{Y}_{ij} \vec{w}_j^T$$

minimizes the average squared Euclidean distance over all vectors. In matrix notation, the approximation over all vectors decomposes as the product of an $N \times k$ matrix and a $k \times d$ matrix as follows.

$$\mathbf{HX} \approx \underbrace{\begin{bmatrix} \mathbf{Y}_{11} & \dots & \mathbf{Y}_{1k} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ \mathbf{Y}_{N1} & \dots & \mathbf{Y}_{Nk} \end{bmatrix}}_{N \times k} \underbrace{\begin{bmatrix} \vec{w}_1^T \\ \vdots \\ \vec{w}_k^T \end{bmatrix}}_{k \times d}.$$

Due to the properties of the scores and loadings, the approximation is a rank k matrix. In the following sections, we'll seek similar decompositions of our data matrix.

4.2.2 SVD and Low Rank Approximations

The standard problem for low rank matrix approximations is to solve the following problem. Given a matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$ and a chosen rank k, we want:

$$\text{argmin}_{\mathbf{Z} \in \mathbb{R}^{N \times d}} \|\mathbf{X} - \mathbf{Z}\|_F^2 =$$

subject to the constraint that $\mathbf{Z}$ has rank k.

Solving this constrained minimization problem may appear difficult, but the answer is obtainable directly from the SVD of $\mathbf{X}$ due to the following theorem.

Theorem 4.3 (Eckart-Young-Mirsky Theorem)**.** *Suppose matrix* $\mathbf{X} \in \mathbb{R}^{N \times d}$ *has singular value decomposition*

$$\mathbf{X} = \mathbf{USV}^T$$

with singular values

$$\sigma_1 \geq \cdots \geq \sigma_{\min\{N,d\}}.$$

Then 1) For any rank k matrix $\mathbf{Z} \in \mathbb{R}^{N \times d}$*,*

$$\|\mathbf{X} - \mathbf{Z}\|_F^2 \geq \sigma_{k+1}^2 + \cdots + \sigma_{\min\{N,d\}}^2$$

2) The rank k matrix attained by keeping the first k left singular vectors, right singular vectors, and singular values of the SVD of $\mathbf{X}$ *attains this minimum. Specifically, if* $\vec{u}_1, \ldots, \vec{u}_k$ *are the first k left singular vectors and* $\vec{v}_1, \ldots, \vec{v}_k$ *are the first k right singular vectors then*

$$\tilde{X} = \sum_{j=1}^{k} \sigma_j \vec{u}_j \vec{v}_j^T$$

is the best rank k approximation to $\mathbf{X}$ *under (squared) Frobenius loss.*

There are several important implications of this theorem. First, the direct result indicates that computing the SVD of $\mathbf{X}$ immediately allows us to compute the best approximation under Frobenius loss for a specified rank k. In practice, the full SVD is not required since we will typically consider the case where $k < \min\{N, d\}$. There is a another implication as well. In cases where a specific choice of k is not clear, the singular values of $\mathbf{X}$ provide a method to comparing different choices of k. Akin to the scree plot, we can plot the (squared) singular values to look for clear separation or alternatively, plot the ratio

$$\frac{\sum_{j=1}^{k} \sigma_j^2}{\sum_{j=1}^{\min\{N,d\}} \sigma_j^2}$$

as a function of k to understand the relative error for a specific choice of k.

For a given choice of k, we now approximate our original data by linear combination of the right singular vectors $\vec{v}_1, \ldots, \vec{v}_k$. The approximations are

$$\vec{x}_i \approx \vec{z}_i = \sum_{j=1}^{k} \sigma_j \mathbf{U}_{ij} \vec{v}_j$$

.

4.2.3 Connections with PCA

Suppose that we were to compute the full SVD of the centered data matrix

$$\mathbf{HX} = \mathbf{USV}^T.$$

We can express the sample covariance matrix of the original data using the SVD as

$$\hat{\Sigma}_X = \frac{1}{N}(\mathbf{HX})^T(\mathbf{HX}) = \frac{1}{N}\mathbf{VS}^T\mathbf{U}^T\mathbf{USV}^T = \mathbf{V}\left(\frac{1}{N}\mathbf{S}^T\mathbf{S}\right)\mathbf{V}^T. \tag{4.7}$$

The matrix $\frac{1}{N}\mathbf{S}^T\mathbf{S} \in \mathbb{R}^{d \times d}$ is diagonal with entries $\sigma_1^2/N \geq \cdots \geq \sigma_d^2/N$. Furthermore, the matrix $\mathbf{V}$ is orthonormal. Thus, from the SVD of $\mathbf{HX}$ we can immediately compute the spectral decomposition of $\hat{\Sigma}_X$ to attain the principal component variances and loadings. In fact, the principal component loadings

are the right singular vectors of **HX** whereas the principal component variances are the squared singular values divided by N, e.g. $\lambda_j = \sigma_j^2/N$. Using this observation,

$$\mathbf{HX} = \mathbf{USV}^T \rightarrow \mathbf{HXV} = \mathbf{US}$$

from which we may conclude the principal component scores are equal to **US**. This connection is the basis for most numerical implementation of PCA since it is more both faster and more numerically stable to compute the SVD of **HX** than to compute both $\hat{\Sigma}_X$ and its eigendecomposition! Thus, computing the best rank k approximation to a centered data matrix is equivalent to the best approximation of the centered data using the first k PC scores.

However, using the SVD to compute a low rank approximation to a *non-centered* data matrix will give a different result than PCA since the SVD of **HX** will be different than the SVD of **X**. Unlike PCA, which decomposes variability in directions relative to the center of the data, SVD learns an orthonormal basis which decomposes variability relative to the origin. Only when the data is centered (so its mean is the origin) do SVD and PCA coincide. Nonetheless, SVD has similar weaknesses to PCA including a sensitivity to scaling and outliers and an inability to detect nonlinear structure.

SVD can provide one final note of insight regarding PCA. Suppose that $N < d$, which is to say that we have fewer samples than the dimensionality of our data. After centering, the matrix **HX** will have rank most $N - 1$. (Centering reduces the maximum possible rank from N to $N-1$). The SVD of **HX** will have at most $d - 1$ non-zero singular values. Thus, $\hat{\Sigma}_X$ will have at most $N - 1$ non-zero PC variances and we can conclude that our data reside on a hyperplane of dimension $N - 1$ (possibly lower if **HX** has rank less than $N - 1$). Since $N - 1 < d$, we are guaranteed to find a lower-dimensional representation of our data! However, this conclusion should be viewed cautiously. Should additional samples be drawn, can we conclude that they would also be constrained to the same hyperplane learned using the first N samples?

4.2.4 Recommender Systems

SVD may also be applied to association rule learning which can identify similar items in a datasets based on partial observations. As a motivating example, consider the case where we have a dataset of user provided ratings of products, which could be items purchased, songs listed to, or movies watched. In this case, $\mathbf{X}_{ij}$ indicates user is rating of item j. Typically, most of the entries of **X** will be NA since users have likely interacted with a small number of items. Using a variant of SVD, a simple recommendation system proceeds in two steps. First, we can impute the missing ratings. We can then use this result to infer similar movies which can be used for recommendation.

4.2.4.1 Imputation

Let $\mathbf{X} \in \mathbb{R}^{N \times d}$ be the data matrix of user ratings with rows corresponding to user and columns to items and

$$\mathcal{I} = \{ij \text{ s.t. } \mathbf{X}_{ij} \neq NA\}$$

be the set of all indices of $\mathbf{X}$ for which we have observed ratings. For any approximating matrix $\tilde{\mathbf{X}} \in \mathbb{R}^{N \times d}$, we may define a 'Frobenius'-like error as

$$\mathbb{L}(\mathbf{X}, \tilde{\mathbf{X}}) = \sum_{ij \in \mathcal{I}} (\mathbf{X}_{ij} - \tilde{\mathbf{X}}_{ij})^2$$

which is the sum-squared error over all observations. Using this definition of loss, here's a simple algorithm for imputing the missing entries of $\mathbf{X}$ using low rank approximations of a pre-specified rank k.

0) We initialize the matrix $\tilde{\mathbf{X}}$ of imputed entries by taking

$$\tilde{X}_{ij} = \begin{cases} X_{ij} & ij \in \mathcal{I} \\ 0 & ij \neq \mathcal{I} \end{cases}.$$

Now let's use the SVD of $\tilde{\mathbf{X}}$ to compute a rank k approximation, $\mathbf{X}^{(k)}$. Update our imputed matrix $\tilde{\mathbf{X}} = \tilde{\mathbf{X}}^{(k)}$ and compute the error

$$\ell = \mathbb{L}(\mathbf{X}, \tilde{\mathbf{X}})$$

. The low-rank approximation will distort all entries of $\tilde{\mathbf{X}}$ so that $\ell > 0$. We now repeat the following two steps.

1) Overwrite the entries of $\tilde{\mathbf{X}}$ corresponding to observations in $\mathbf{X}$, e.g. for all $ij \in \mathcal{I}$, set $\tilde{\mathbf{X}}_{ij} = \mathbf{X}_{ij}$. The entries corresponding to missing observations generated by the low-rank approximation are kept unchanged. Now recompute the SVD of $\tilde{\mathbf{X}}$ to find the rank k approximating matrix $\tilde{\mathbf{X}}^{(k)}$. Update our imputed matrix using the low-rank approximation so that $\tilde{\mathbf{X}} = \mathbf{X}^{(k)}$ and recompute the error $\ell^* = \mathbb{L}(\mathbf{X}, \tilde{\mathbf{X}})$.

2) If $\ell^* < \ell$ and $|\ell - \ell^*|/\ell > \epsilon$ then set $\ell = \ell^*$ and return to step (1). Else stop the algorithm and we use matrix $\tilde{\mathbf{X}}$ as our matrix of imputed values.

In summary, after initialization, we are continually overwriting the entries of our matrix of imputed values corresponding to observations then applyin a low-rank approximation. We stop the algorithm when the error stops decreasing or when the relative decrease in error is less than a specified threshhold ϵ. In addition to the rank k and the stopping threshhold ϵ there is one other important 'tuning' parameter, the initialization. In the brief description above, we used a standard of 0, but one could also use the average of all entries in the corresponding column (average item rating) or row (average rating given by a user) or some other specification. Many more complicated recommendation systems include user

and item specific initializations and adjustments but still imploy a low rank approximation somewhere in their deployment.

In later updates, we show an application of this algorithm to the Movielens 10m dataset containing 10 million ratings from 72,000 viewers for 10,000 movies. To handle the size of this dataset, we use the Fast Truncated Singular Value Decomposition.

```
# library("irlba")
# initialize <- function(mat){
# # get column means ignoring NAs
# ave.rat <- colMeans(mat,na.rm = TRUE)
# # fill NAs by average movie rating
# for(j in 1:ncol(mat)){
#   mat[is.na(mat[,j]),j] <- ave.rat[j]
# }
# return(mat)
# }
#
# maxim <- function(mat,k){
# # temp <- svd(mat)
# temp<- irlba(mat, nv = k)
# return(list(U = temp$u[,1:k],
#             D = temp$d[1:k],
#             V = temp$v[,1:k],
#             mat.hat = temp$u[,1:k] %*% diag(temp$d[1:k]) %*% t(temp$v[,1:k])))
# }
#
# recommender <- function(mat, num_steps, k){
# # initialize loss function tracking
# loss <- rep(NA,num_steps)
# # run EM algorithm and save loss
# ind.known <- !is.na(mat)
# mat2 <- initialize(mat)
# for (j in 1:num_steps){
#   mat2 <- maxim(mat2,k)$mat.hat
#   loss[j] <- sum((mat2[ind.known] - mat[ind.known])^2)
#   mat2[ind.known] <- mat[ind.known]
# }
# return(list(loss= loss, fit = mat2))
# }
#
# k <- 3
# temp <- recommender(ratings,num_steps = 200, k = k)
# plot(temp$loss, xlab = "Step", ylab = expression(ell))
```

4.2.4.2 Recommendation

Suppose now that we have a matrix, $\tilde{\mathbf{X}} \in \mathbb{R}^{N\times d}$ of movie ratings (real or imputed) and its SVD

$$\tilde{\mathbf{X}} = \tilde{\mathbf{U}}\tilde{\mathbf{S}}\tilde{\mathbf{V}}^T$$

where $\tilde{\mathbf{U}} \in \mathbb{R}^{N\times k}$, $\tilde{\mathbf{S}} \in \mathbb{R}^{k\times k}$ and $\tilde{\mathbf{V}} \in \mathbb{R}^{d\times k}$. Then for user i the rating they give to movie j is a linear combination of the elements in the j column of $\tilde{\mathbf{V}}^T$. Specifically,

$$\tilde{\mathbf{X}} \approx \sum_{\ell=1}^{k} \sigma_\ell \tilde{\mathbf{U}}_{i\ell}(\tilde{\mathbf{V}})^T_{\ell j} = \sum_{\ell=1}^{k} \sigma_\ell \tilde{\mathbf{U}}_{i\ell}\tilde{\mathbf{V}}_{j\ell}.$$

For any movie, its rating will always be a linear combination of the elements in the corresponding column of $\tilde{V}^T$. As such, we may view the k-dimensional vectors in each column of $\tilde{\mathbf{V}}$ as a representation of that movie. We may then use these vectors to identify similar movies; one common approach is the cosine similarity, which for vectors $\vec{x}, \vec{y} \in \mathbb{R}^k$ is the cosine of the angle between them, i.e.

$$\cos\theta = \frac{\vec{x}^T\vec{y}}{\|\vec{x}\|\|\vec{y}\|}.$$

The cosine similarity is bounded between -1 and 1 and two vectors are considered more similar if the cosine of the angle between them is closer to 1. Using this representation, we can take any movie (a column of $\tilde{\mathbf{V}}^T$) and choose the most similar movie (choosing the other columns of $\tilde{\mathbf{V}}^T$ with the largest cosine similarity). Thus, if a user gave a high rating to movie b, we now have a method for recommending one or more similar movies they might enjoy.

4.3 Nonnegative Matrix Factorization

In both PCA and SVD, we learn data-driven orthonormal feature vectors which we can use to decompose our data in an orderly fashion. Nonnegative matrix factorization (NMF) is again focused on learning a set latent vectors which can be used to approximate our data. However, we will add a few restrictions motivated by experimental data and a desire to increase interpretability of the results which will drastically alter the results.

For NMF, we focus on cases where $\mathbf{X}$ is an $N \times d$ data matrix with the added condition that its entries are nonnegative. Notationally, we write $\mathbf{X} \in \mathbb{R}^{N\times d}_{\geq 0}$ to indicate it is composed of nonnegative real values. The nonnegativity condition is a natural constraint for many experimental data sets, but we are also going to impose a similar constraint on our feature vectors and coefficients. More specifically, for a specific rank k, we seek a coefficient matrix $\mathbf{W} \in \mathbb{R}^{N\times k}_{\geq 0}$ and feature matrix $\mathbf{H} \in \mathbb{R}^{k\times d}_{\geq 0}$ such that $\mathbf{WH}$ is as close to $\mathbf{X}$ as possible. The nonnegativity constraint on the elements of $\mathbf{W}$ and $\mathbf{H}$ implies that $\mathbf{WH} \in \mathbb{R}^{N\times d}_{\geq 0}$. There are many different measures of proximity that one may use in NMF which

are greater than zero and equal to zero if and only if $\mathbf{X} = \mathbf{WH}$. The most common measures are

- Frobenius norm: $\|\mathbf{X} - \mathbf{WH}\|_F$.
- Divergence: $D(\mathbf{X}\|\mathbf{WH}) = \sum_{i=1}^{N}\sum_{j=1}^{d}\left[\mathbf{X}_{ij}\log\frac{\mathbf{X}_{ij}}{(\mathbf{WH})_{ij}} + (\mathbf{WH})_{ij} - \mathbf{X}_{ij}\right]$
- Itakura-Saito Divergence:

$$D_{IS}(\mathbf{X}, \mathbf{WH}) = \sum_{i=1}^{N}\sum_{j=1}^{d}\left[\frac{\mathbf{X}_{ij}}{(\mathbf{WH})_{ij}} - \log\frac{\mathbf{X}_{ij}}{(\mathbf{WH})_{ij}} - 1\right]$$

These loss functions emphasize and prioritize different features of the data and are often coupled with additional penalties or assumptions on the elements of $\mathbf{W}$ and $\mathbf{H}$ which we will discuss at the end of the section. From a statistical perspective, they may also be motivated based on differing beliefs about the data generating process. For example, suppose we assume conditional independence of the entries of $\mathbf{X}$ given $\mathbf{W}$, $\mathbf{H}$, and error variance σ. Under a Gaussian error model

$$\mathbf{X}_{ij} \mid \mathbf{W}, \mathbf{H} \sim \mathcal{N}\left((\mathbf{WH})_{ij}, \sigma^2\right)$$

maximizing the associated likelihood w.r.t. $\mathbf{W}$ and $\mathbf{H}$ is equivalent to minimization of the Frobenius norm $\|\mathbf{X} - \mathbf{WH}\|_F^2$. We'll revisit this idea in greater detail in Section 4.3.6.

For now, let us focus on the primary motivation of NMF, which is to create more interpretable feature vectors (the rows of $\mathbf{H}$).

4.3.1 Interpretability, Superpositions, and Positive Spans

Consider the case of SVD where we can approximate a given vector $\vec{x}_i$ using the first k right singular vectors as

$$\vec{x}_i \approx \sum_{j=1}^{k} \sigma_j \mathbf{U}_{ij}\vec{v}_j.$$

Let us restrict ourselves to the case of nonnegative entries. Suppose for the moment that the ℓth coordinate of $\vec{x}_i$ is (very close to) zero and this is reflected by our approximation as well so that

$$(\vec{x}_i)_\ell \approx \sum_{j=1}^{k} \sigma_j \mathbf{U}_{ij}(\vec{v}_j)_\ell \approx 0.$$

The freedom of the coefficients $\sigma_j\mathbf{U}_{ij}$ and the features $(\vec{v}_j)_\ell$ for $j = 1, \ldots, k$ to take any value in $\mathbb{R}$ prevents us from concluding that there is a comparable `near-zeroness` in the features or coefficients. In could be that case that $\mathbf{U}_{ij}(\vec{v}_j)_\ell$ is near zero for all $j = 1, \ldots, k$ or that some subset are large and positive but

are canceled out by a different subset that is large and negative. If, however, we restrict the coefficients and features to be zero this cancellation effect cannot occur. Features can only then be added but never subtracted. Under this restriction, $\sum_{j=1}^{k} \sigma_j \mathbf{U}_{ij}(\vec{v}_j)_\ell$ will only be close to zero if the coefficients are (near) zero for any feature vector which has a (large) positive entry in its ℓth coordinate.

Thus, in a factorization of the form

$$\vec{x}_i \approx \sum_{j=1}^{k} \underbrace{\mathbf{W}_{ij}}_{\geq 0} \underbrace{\vec{h}_j}_{\in \mathbb{R}^d_{\geq}},$$

we can only superimpose (add) features to approximate our data, we might expect the features themselves to look more like our data. In matrix notation, we have

$$\mathbf{X} = \mathbf{WH}$$

where the coefficients for each sample are stored in the rows of $\mathbf{W} \in \mathbb{R}^{N \times k}_{\geq 0}$ and the features are transposed and listed as the rows of $\mathbf{H} \in \mathbb{R}^{k \times d}$.

4.3.2 Geometric Interpretation

As a motivating example consider the case, where we have data $\vec{x}_1, \ldots, \vec{x}_N \in \mathbb{R}^3_{\geq 0}$ for which there is a exact decomposition

$$\mathbf{X} = \mathbf{WH}$$

which is to say there are nonnegative coefficients, $\mathbf{W}$ and feature vectors $\vec{h}_1, \vec{h}_2 \in \mathbb{R}^3_{\geq 0}$ such that

$$\vec{x}_i = \mathbf{W}_{i1}\vec{h}_1 + \mathbf{W}_{i2}\vec{h}_2 \qquad \text{for } i = 1, \ldots, N.$$

The nonnegativity condition on data implies that $\vec{x}_1, \ldots, \vec{x}_N$ reside in the positive orthant of $\mathbb{R}^3$. The exact decomposition assumptions implies $\{\vec{x}_1, \ldots, \vec{x}_N\} \in \text{span}\{\vec{h}_1, \vec{h}_2\}$ and furthermore the following more restricted picture holds.

The data are constrained within the positive span of the vectors, a notion we may now define.

Definition 4.2 (Positive Span)**.** The positive span of a set of vectors $\{\vec{h}_1, \ldots, \vec{h}_k\} \in \mathbb{R}^d$ is the set

$$\Gamma\left(\{\vec{h}_1, \ldots, \vec{h}_k\}\right) = \left\{ \vec{v} \in \mathbb{R}^d \,\middle|\, \vec{v} = \sum_{j=1}^{k} a_j \vec{h}_j,\ a_1, \ldots, a_k \geq 0 \right\}.$$

This set is also called the simplicial cone or conical hull of $\{\vec{h}_1, \ldots, \vec{h}_k\}$.

In the motivating example above, our data live in the positive span of the two feature vectors, thus we say the data matrix $\mathbf{X}$ has a nonnegative matrix factorization $\mathbf{WH}$.

Thus, we may view NMF as a restricted version of PCA or SVD where we have moved from spans to positive spans. This seemingly small change gives rise to some big problems. Even in this simple case above we have a uniqueness problem. Up to sign changes and ordering, the feature vectors in PCA and SVD were unique. However, we can find two alternative vectors $\vec{h}_1'$ and $\vec{h}_2'$ which still give a exact NMF. There are trivial cases. First, one can change ordering ($\vec{h}_1' = \vec{h}_2$ and $\vec{h}_2' = \vec{h}_1$) which we avoid in PCA and SVD by assuming the corresponding singular values of PC variances are listed in decreasaing order. Secondly, we could rescale by setting $\vec{h}_1' = c\vec{h}_1$ and rescaling the corresponding coefficients by a factor of $1/c$ for some $c > 0$ which PCA and SVD avoid by fixing feature vectors to have unit length. The ordering issue is a labeling concern and may be

ignored, whereas the rescaling issue can be addressed by adding constraints on the length of the feature vectors.

A third and far more subtle issue occurs because we do not enforce orthogonality. In @ref[fig:nmf-ex], imagine that the feature vectors are the arms of a folding fan. We could change the angle between our feature vectors by opening or closing the arms of the fan. So long as we do not close the fan too much (and leave our data outside the positive span) or open them too much (so that feature vectors have negative entries), we can still find a perfect reconstruction. This 'folding fan' issue can be addressed through additional penalties which we discuss in @ref{sec-nmf-ext}, but nonuniqueness cannot be eliminated entirely. Thus, we seek **an** NMF for our data rather than **the** NMF.

4.3.3 Finding a NMF: Multiplicative Updates

For a given choice of error, the lack of a unique solution also means there is no closed form solution for $\mathbf{W}$ and $\mathbf{H}$. Thus, we will need to apply numerical optimization to find a $\mathbf{W}$ and $\mathbf{H}$ which minimizes the selected error,

$$\text{argmin}_{\mathbf{W}\in\mathbb{R}^{N\times k}_{\geq 0},\mathbf{H}\in\mathbb{R}^{k\times d}_{\geq 0}} \quad D(\mathbf{X},\mathbf{WH})$$

. The loss is a function of all of the entries of $\mathbf{W}$ and $\mathbf{H}$. To apply any type of gradient based optimization, we need to compute the partial derivative of our loss with respect to each of the entries of these matrices.

As an example, we will focus on the divergence and show the relevant details. For gradient based optimization, we then need to compute $\frac{\partial}{\partial \mathbf{W}_{ij}} D(\mathbf{X}\|\mathbf{WH})$ for ' $\leq i \leq N$ and $1 \leq j \leq k$ and $\frac{\partial}{\partial \mathbf{H}_{j\ell}} D(\mathbf{X}\|\mathbf{WH})$ for $1 \leq j \leq k$ and $1 \leq \ell \leq d$. Note that

$$(\mathbf{WH})_{st} = \sum_{j=1}^{k} \mathbf{W}_{sk}\mathbf{H}_{kt}$$

so that

$$\frac{\partial (\mathbf{WH})_{st}}{\partial (\mathbf{WH})_{ij}} = \begin{cases} \mathbf{H}_{jt} & s = i \\ 0 & s \neq i \end{cases}$$

Thus, we may may make use of the chain rule to conclude that

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{W}_{ij}} D(bfX\|\mathbf{WH}) &= \frac{\partial}{\partial \mathbf{W}_{ij}} \sum_{st} \left(\mathbf{X}_{st}\log((\mathbf{X})_{st}) - \mathbf{X}_{st}\log((\mathbf{WH})_{st}) + (\mathbf{WH})_s t - (\mathbf{X})_{st}\right) \\
&= \frac{\partial}{\partial \mathbf{W}_{ij}} \sum_{st} \left(-\mathbf{X}_{st}\log((\mathbf{WH})_{st}) + (\mathbf{WH})_{st}\right) \\
&= \sum_{st} \left(\frac{\mathbf{X}_{st}}{(\mathbf{WH}_{st})} + 1\right) \frac{\partial (\mathbf{WH})_{st}}{\partial \mathbf{W}_{ij}} \\
&= \sum_{t=1}^{d} \left(-\frac{\mathbf{X}_{it}}{(\mathbf{WH}_{it})} + 1\right) \mathbf{H}_{jt}.
\end{aligned}$$

A similar calculation gives

$$\frac{\partial}{\partial \mathbf{H}_{ij}} D(bfX \| \mathbf{WH}) = \sum_{s=1}^{N} \left(-\frac{\mathbf{X}_{sj}}{(\mathbf{WH})_{sj}} + 1 \right) \mathbf{W}_{sj}.$$

In a standard implementation of gradient descent, we choose a step size $\epsilon > 0$ and apply the updates

$$\begin{aligned} \mathbf{W}_{ij} &\leftarrow \mathbf{W}_{ij} - \epsilon \sum_{t=1}^{d} \left(-\frac{\mathbf{X}_{it}}{(\mathbf{WH}_{it})} + 1 \right) \mathbf{H}_{jt} \\ \mathbf{H}_{ij} &\leftarrow \mathbf{H}_{ij} - \epsilon \sum_{s=1}^{N} \left(-\frac{\mathbf{X}_{sj}}{(\mathbf{WH})_{sj}} + 1 \right) \mathbf{W}_{sj} \end{aligned} \tag{4.8}$$

to each entry of $\mathbf{W}$ and $\mathbf{H}$ simultaneously. Alternatively, we could consider coordinate descent where we update each entry of $\mathbf{W}$ (holding all other entries of $\mathbf{W}$ and $\mathbf{H}$ constant) separately then do the same for $\mathbf{H}$ then repeat. Each approach will converge to a local mode (though possibly different ones) when ϵ is sufficiently small. However, if ϵ is too small it may take many iterations to converge. Unfortunately, choosing ϵ can create a numerically unstable algorithm (that doesn't converge at all) or one where we lose the nonnegativity condition on the entries of $\mathbf{W}$ or $\mathbf{H}$.

To preserve nonnegativity, [**?**] developed a set of state dependent step-sizes (one for each entry of $\mathbf{W}$ and $\mathbf{H}$) which result in multiplicative rather than additive updates. For divergence, they take step sizes

$$\epsilon_{ij}^{W} = \frac{\mathbf{W}_{ij}}{\sum_t \mathbf{H}_{jt}} \text{ and } \epsilon_{ij}^{H} = \frac{\mathbf{H}_{ij}}{\sum_s \mathbf{W}_{sj}}.$$

These step size are proportional to the entries we are updating so that we take larger steps for larger entries of $\mathbf{W}$ of $\mathbf{H}$. If we substitute these step sizes in the updates simplify to

$$\begin{aligned} \mathbf{W}_{ij} &\leftarrow \mathbf{W}_{ij} \left[\frac{\sum_t (\mathbf{H}_{jt} \mathbf{X}_{it} / (\mathbf{WH}_{it}))}{\sum_t \mathbf{H}_{jt}} \right] \\ \mathbf{H}_{ij} &\leftarrow \mathbf{H}_{ij} \left[\frac{\sum_s (\mathbf{W}_{si} \mathbf{X}_{\mathbf{sj}} / (\mathbf{WH})_{si})}{\sum_s \mathbf{W}_{si}} \right] \end{aligned} \tag{4.9}$$

indicating that we rescale the entries of $\mathbf{W}$ and $\mathbf{H}$ by some nonnegative value which is guaranteed to preserve the nonnegativity of each entry. A short proof showing convergence to a local mode may be found in the original paper by Lee and Seung along with multiplicative update rules for other choices of loss are also available in [**?**]. Deriving these updates are also provided as exercises for the reader. Furthermore, one can verify that the multiplicative updates simplify to one when $\mathbf{X} = \mathbf{WH}$.

We can more compactly write the updates for each matrix using Hadamard notation, which gives the following multiplicative updates for $\mathbf{W}$ and $\mathbf{H}$

$$\begin{aligned} \mathbf{W} &\leftarrow \mathbf{W} \odot \left\{ [\mathbf{X} \oslash (\mathbf{WH})]\mathbf{H}^T \right\} \oslash [\mathbf{1}_{N\times d}\mathbf{H}^T] \\ \mathbf{H} &\leftarrow \mathbf{H} \odot \left\{ \mathbf{W}^T[\mathbf{X} \oslash (\mathbf{WH})] \right\} \oslash [\mathbf{W}^T\mathbf{1}_{N\times d}] \end{aligned} \tag{4.10}$$

In most implementations of NMF, multiplicative updates for $\mathbf{W}$ and $\mathbf{H}$ are alternated until convergence. However, some packages also implement rescaling following each pair of multiplicative updates to enforce other constraints (such as sum to one constrains on the rows of $\mathbf{W}$ or columns of $\mathbf{H}$). When using NMF it is good to check documentation to see if these additional steps are implemented.

4.3.4 NMF in practice

For a given choice of loss, standard NMF proceeds in the following manner.

1) Choose a rank k and initial $\mathbf{W}_{\in}\mathbb{R}^{N\times k}_{\geq 0}$ and $\mathbf{H} \in \mathbb{R}^{k\times d}_{\geq 0}$.
2) Apply the multiplicative rule until a local minimum is reached.
3) Repeats steps (1) and (2) for different initial conditions then select the final $\mathbf{W}$ and $\mathbf{H}$ which give the lowest overall loss.

Different initial conditions (ICs) will converge to different local modes; reptition over multiple differents (ICs) will help find a better overall minimizer and avoid getting a final NMF which is trapped in a poor local mode. There is no guidance of how many ICs. Many packages default to five. More would be better but computational resources are not infinite, and finding even one mode though coordinate ascent can be slow. There are also methods based on nonnegativity constraints of the SVD (aptly named nonnegative SVD or NNSVD) for initialization which have shown good performance [**?**]. However, NNSVD initialization does increase computation burden.

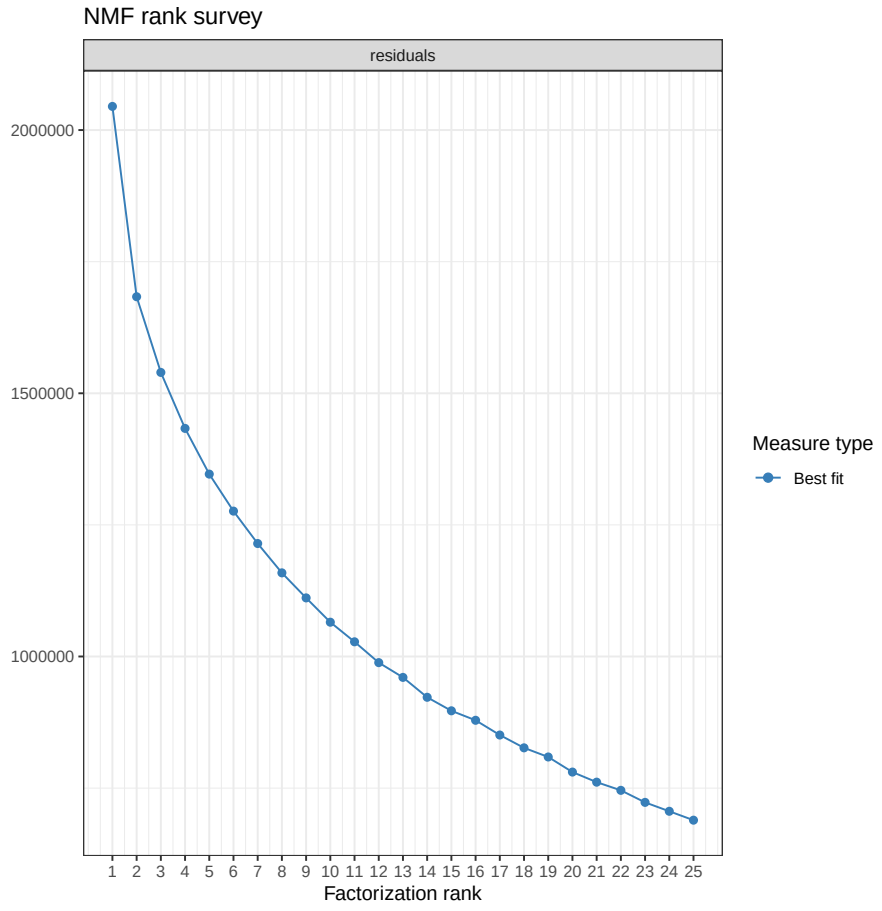
The choice of k is also challenging. If prior information or other project constraints dictate a specific choice of k one should use that value. Alternatively, if visualization is a priority taking $k = 1, 2$, or 3 is a natural choice. Unfortunately, in many cases prior information is typically absent, thus data driven methods for selecting k from a range of possible choices are required. Importantly, since there are no connections between the rank k NMF and rank $k + 1$ NMF, one cannot truncate the vectors and coefficients and attain an optimal solution for a lower-dimensional representation as we can in PCA or SVD. Thus, separate fits at each choice of k must be attained through separate runs of the numerical optimization making NMF far more computationally demanding in practice.

Suppose for now, we have optimal NMF fits for a range of nonnegative ranks obtained by fitting separate NMF models for a range of ranks using multiple initial conditions or NNSVD based initialization at each rank. Given these fits, one natural approach is to follow a `scree plot` like approach similar to what we

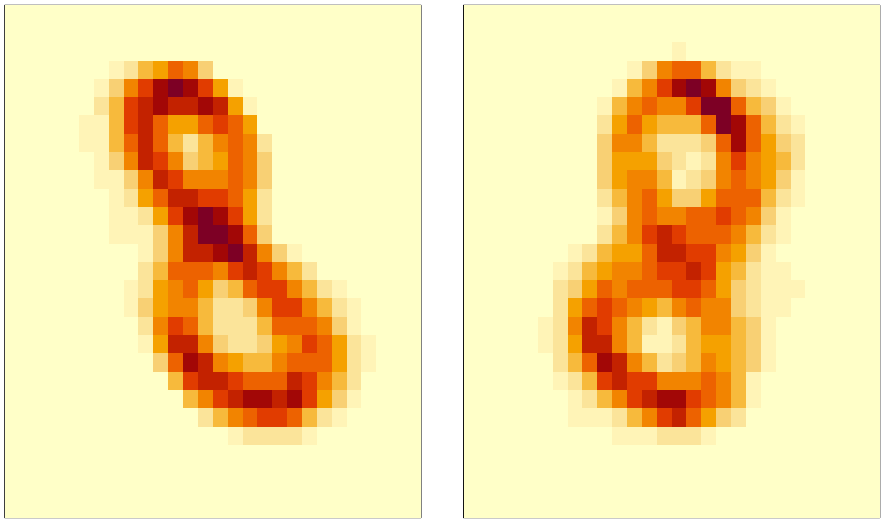
have used for PCA and NMF. One can then compare plot the loss as a function of k and look for an elbow where the error appears to saturate.

Much like SVD and PCA, unfortunately, there is often no clear elbow indicating an obvious choice of k. As such, additional approaches have been suggested in the literature such as cross-validation where in the data is split into test-train sets and the robustness of loss is investigated at each rank. Bayesian methods for balancing low loss with model complexity (small k) are also viable but again rely on some prior on model complexity.Other authors have used connections between NMF and clustering to adapt methods for assessing the quality of (hierarchical) clustering to the case of finding an optimal k in NMF. For example, one can use the robustness (or lack thereof) of clustering of data at each rank measured by the cophenetic correlation to identify an optimal k [**?**]. We'll discuss hierarchical clustering and the cophenetic correlation in more detail Chapter 7. For now, we'll restate the advice from [**?**] and advocate for the value of k at which the cophenetic correlation begins to drop.

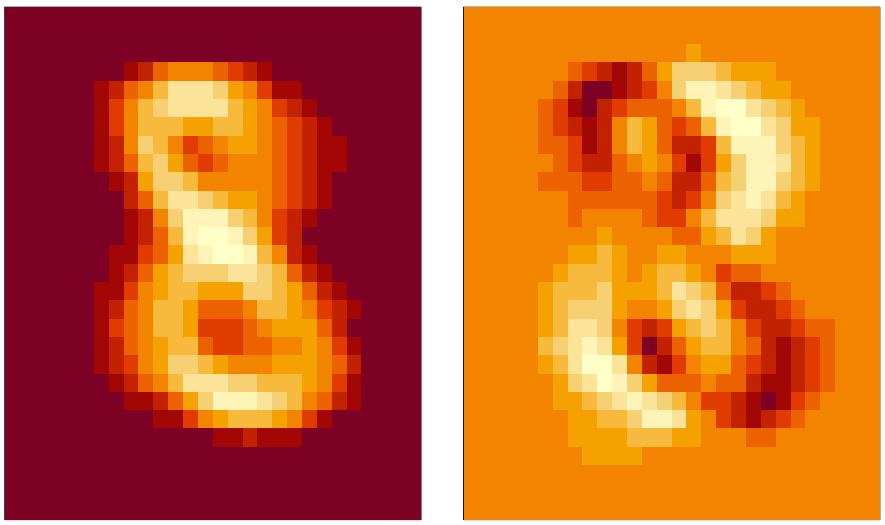
Example 4.2 (NMF applied to MNIST)**.** As an example, we show NMF applied to the first 100 eights (to save computation time) from the MNIST dataset [**?**] using Divergence with five runs per rank. Below we fit separate NMFs for ranks 1 through 25 with 5 random initializations per fit. Divergence (residuals) and the cophenetic correlation are each shown below as a function of rank.



There is no clear elbow in the plot of the model residuals that clearly suggests an optimal rank between 1 and 25. The cophenetic correlation begins to drop after rank 2. Thus, the method of [**?**] suggests $k = 2$ is reasonable. In fact if we select a rank 2 NMF, we do get interpretable 8-like images as our feature images(vectors) which correspond to eights slanting in two different directions, a seemingly reasonable compression of structure in the data.



Compare this to the less 8-like images found using the features given by a rank 2 approximation from SVD.



4.3.5 Regularization and Interpretability

For a random initialization, the values of $\mathbf{W}$ and $\mathbf{H}$ obtained when the multiplicative rules are run to convergence not resemble the original data, which is contrary to our original goal of finding features which are more comparable to the data. More specifically, the feature vectors $\vec{h}_1, \ldots, \vec{h}_k$ which are super-

imposed to approximate the observations may appear quite different from the observations themselves. Many extensions of NMF address this issue through the inclusion of additional penalties on the elements of **W** or **H** which induce sparsity in the coefficients **W** and/or constrain the features to be more similar to the data. In practice, it is insufficient to apply penalties which depend solely on the scaling of either **W** or **H** as these can typically be made arbitrarily small by increasing corresponding elements of **H** or **W** respectively. Thus, any penalty which depends on the scaling of one matrix often includes additional constraints or penalties on the other.

Briefly, we discuss two version which are constructed to generate features which strongly mimic the original observations.

4.3.5.1 Archetypes

Before discussing Archetypal Analysis, we need an additional geometric idea.

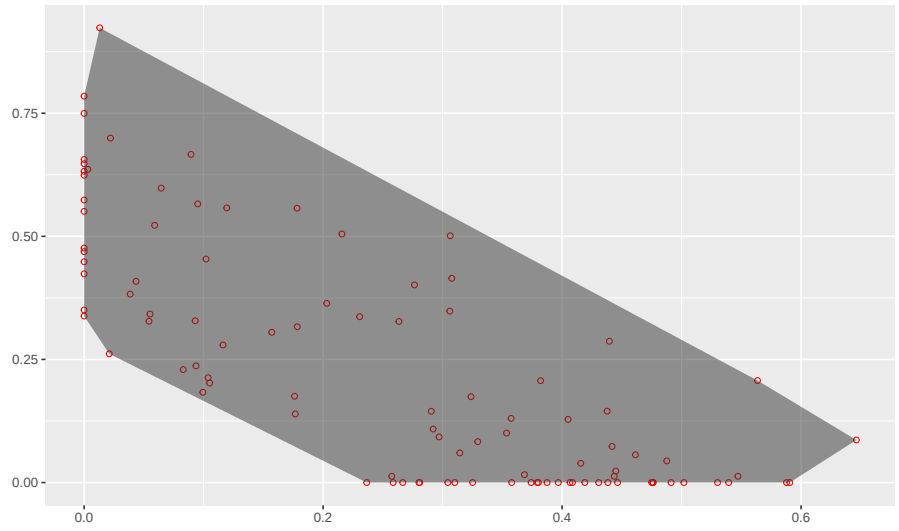
Given a set of vectors $\mathcal{H} = \{\vec{h}_1, \ldots, \vec{h}_k\} \subset \mathbb{R}^d$, the convex hull of $\mathcal{H}$ is the set

$$\text{conv}(\mathcal{H}) = \{b_1\vec{h}_1 + \cdots + b_k\vec{h}_k : b_1, \ldots, b_k \geq 0,\, c_1 + \cdots + c_k = 1\}$$

The weights in the above definition are constrained to be positive, similar to the definition of positive span, but with the added constraint that they sum to one. Thus, for a set of vectors $\mathcal{H}$

$$\text{conv}(\mathcal{H}) \subset \Gamma(\mathcal{H}) \subset \text{span}(\mathcal{H}).$$

An example of a convex hull (shown in grey) of set of vectors (points in red) in $\mathbb{R}^2$ is shown below. Imagine wrapping a string around the set of points and pulling it tight. Everything on/within the loop is in the convex hull. An analogous interpretation holds in higher dimensions using a "sheet" rather than a string.

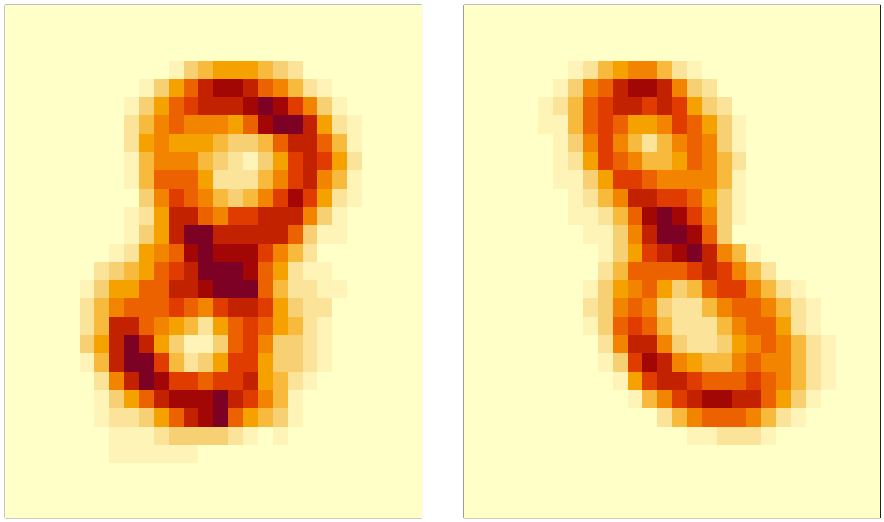


Importantly, we can view the coefficient $c_1, \ldots, c_k$ as defining (probability) weights. Thus, every point in the convex hull of $\mathbf{H}$ is a weighted average of the vectors in $\mathcal{H}$ and one then expects any point in the convex hull of $\mathcal{H}$ to resemble the vectors in $\mathcal{H}$. In archetypal analysis [**?**], we add the constraint that the features (row of $\mathbf{H}$) are themselves elements in the convex hull of the original data so that $\mathbf{H} = \mathbf{BX}$ where $\mathbf{B} \in \mathbb{R}_{\geq 0}^{k \times N}$ must satisfy the constraint $\mathbf{B}\vec{1}_N = \vec{1}_k$.

Under this setup, we then want to solve the following constrained optimization problem

$$\text{argmin}_{\mathbf{W} \in \mathbb{R}_{\geq 0}^{N \times k}, \mathbf{B} \in \mathbb{R}_{\geq 0}^{k \times N}, \mathbf{B}\vec{1}_N = \vec{1}_k} \|\mathbf{X} - \mathbf{WBX}\|_F.$$

The rows of $\mathbf{BX}$, called archetypes, reside are typically extremal in the sense that they reside on the boundary of the convex hull of the data. We show them below for a rank two approximation to the eights consistent with the choice in the example above.



4.3.5.2 Volume Regularized NMF

Much like Archetype Analysis, Volume regularized NMF creates feature vectors which resemble the observation through a more geometric approach [**?**, **?**]. Suppose that we are seeking a rank k approximation of data in d dimensions with $k < d$. If the vectors $\vec{h}_1, \ldots, \vec{h}_k$ are affinely independent, i.e. they don't live on a hyperplane of dimension less than k, then the volume of the convex of hall of the vectors is given by the following formula

$$\text{vol}\left(\text{conv}(\{\vec{h}_1, \ldots, \vec{h}_k\})\right) = \frac{\sqrt{\det(\tilde{\mathbf{H}}\tilde{\mathbf{H}}^T)}}{(k-1)!}$$

where

$$\tilde{\mathbf{H}} = \begin{bmatrix} \vec{h}_1^T - \vec{h}_k^T \\ \vdots \\ \vec{h}_{k-1}^T - \vec{h}_k^T \end{bmatrix} \in \mathbb{R}^{(k-1)\times d}.$$

An extension of this result is that for linearly independent vectors $\vec{h}_1, \ldots, \vec{h}_k$, the volume of the convex hull of the vectors $\vec{0}, \vec{h}_1, \ldots, \vec{h}_k$ can be computed directly from our feature matrix $\mathbf{H}$, i.e.

$$\text{vol}\left(\text{conv}(\{0, \vec{h}_1, \ldots, \vec{h}_k\})\right) = \frac{\sqrt{\det(\mathbf{H}\mathbf{H}^T)}}{k!}.$$

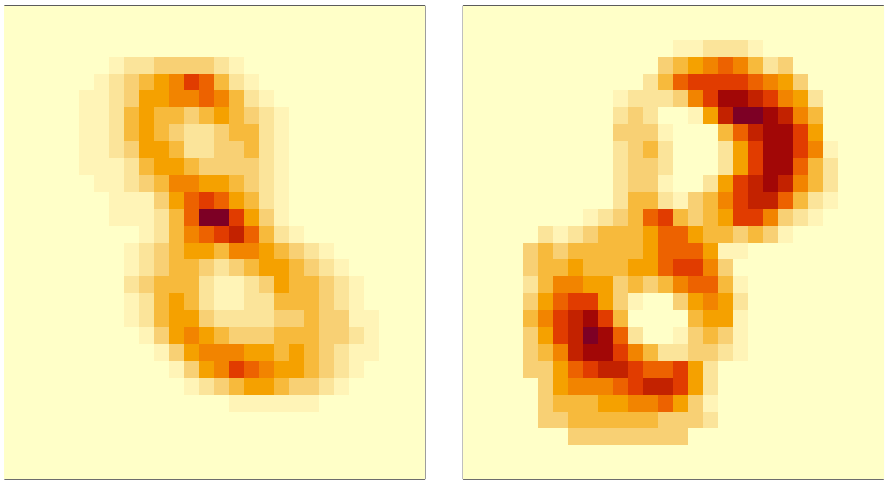
One brief aside on the notion of volume here that is not typically made clear in the related literature. When $k < d$, the convex hull of $\vec{0}, \vec{h}_1, \ldots, \vec{h}_k$ is contained in a k-dimensional hyperplane. Thus, its d-dimensional volume must be zero.

When we talk of the volume of a convex hull, we instead mean its k-dimensional volume. As an example, consider the case of a single vector $\vec{h}_1$ in $\mathbb{R}^3$. The convex hull of $\vec{0}, \vec{h_1}$ is the line segment from the origin to $\vec{h}_1$. Line segments have zero 3-dimensional volume, but they do have length (1-dimensional volume). In this case, the 1-dimensional volume of the convex hull is the length of the line segment, e.g. $\|\vec{h}_1\|$.

Incorporating a penalty on the volume of the convex hull of the feature vectors gives rise to the aptly named volume-regularized NMF

$$\text{argmin}_{\mathbf{W}\in\mathbb{R}^{N\times k}_{\geq 0}, \mathbf{W}\vec{1}_N=\vec{1}_k, \mathbf{H}\in\mathbb{R}^{k\times d}_{\geq 0}} \quad \|\mathbf{X}-\mathbf{WH}\|_F^2 + \lambda\sqrt{\det(\mathbf{HH}^T)}$$

here the constraint $\mathbf{W}\vec{1}_N = \vec{1}_k$ requiring that the rows of $\mathbf{W}$ sum to one prevents an arbitrary shrinkage of the feature vectors towards zero. In some numerical implementation, this constraint is relaxed so that the sum of the rows of $\mathbf{W}$ are bounded above by one. The value λ is tuning parameter that must be specified by the user to balance goodness of fit vs minimal volume. Below we show the results of volume regularized NMF applied to the subset of the eights data studied above which strongly resemble results from the standard NMF implementation. However, this method was far more computationally efficient. Only one run of the VR-NMF was required, and it converged much more quickly than the vanilla NMF.



4.3.6 NMF and Maximum Likelihood Estimation

Throughout this section we have focused on the geometric aspects of NMF, but there are parallel statistical perspectives based on maximum likelihood estimation. For example, consider the model

$$\mathbf{X}_{ij} \mid (\mathbf{WH}_{ij}) \sim \mathcal{N}\left((\mathbf{WH}_{ij}, \sigma^2\right), \qquad 1 \leq i \leq N, 1 \leq j \leq d$$

with the additional assumption that the observations in $\mathbf{X}$ are conditionally independent given $\mathbf{WH} \in \mathbb{R}^{N\times k}_{\geq 0}$, $\mathbf{H} \in \mathbb{R}^{k\times d}_{\geq}0$ and σ. Under this framework the likelihood of the data is then

$$\mathcal{L}(\mathbf{X} \mid \mathbf{WH}) \propto \frac{1}{\sqrt{2\pi\sigma^2}} \prod_{i=1}^{N}\prod_{j=1}^{d} \exp\left(-\frac{(\mathbf{X}_{ij} - (\mathbf{WH}_{ij}))^2}{\sigma^2}\right)$$

which has log-likelihood

$$\log \mathcal{L}(\mathbf{X} \mid \mathbf{WH}) = -\frac{N}{2}\log\sigma^2 - \frac{1}{\sigma^2}\sum_{i=1}^{N}\sum_{j=1}^{d}(\mathbf{X}_{ij} - (\mathbf{WH}_{ij}))^2 = -\frac{N}{2}\log\sigma^2 - \frac{1}{\sigma^2}\|\mathbf{X}-\mathbf{WH}\|_F^2.$$

Thus, minimizing the Frobenius norm is equivalent to maximizing the likelihood in this normal model with homoskedastic errors.

Similarly, divergence arises from a Poisson model

$$\mathbf{X}_{ij} \mid (\mathbf{WH})_{ij} \sim \text{Pois}\left((\mathbf{WH})_{ij}\right)$$

again with a low rank structure for $\mathbf{W}$ and $\mathbf{H}$ and the conditional independence assumption on the entries of $\mathbf{X}$ given $\mathbf{WH}$.

The IS divergence again uses conditional independence and a low rank structure for $\mathbf{WH}$ under the assumption of multiplicative Gamma distributed errors with mean one, i.e.

$$\mathbf{X}_{ij} \mid (\mathbf{WH})_{ij} = (\mathbf{WH})_{ij}e_{ij}, \qquad e_{ij} \sim_{iid} \text{Gamma}(\alpha, \beta)$$

with $\alpha/\beta = 1$. See [**?**] for additional details.

4.4 Multidimensional Scaling

Multidimensional scaling (MDS) is a broad name for a myriad of different methods which are designed to handle a common problem. Suppose there are N objects of interest in a dataset. Examples include a set of geographic locations or cells from mass spectrometry or census blocks, etc. Importantly, we do not need actual data corresponding to each object. In MDS, we instead require a measure the distance or dissimilarity between each pair of objects. We can organize these distances into a matrix $\mathbf{\Delta} \in \mathbb{R}^{N\times N}$, with $\mathbf{\Delta}_{rs}$ representing the distance/dissimilarity between objects r and s. Thus, $\mathbf{\Delta}_{rr} = 0$ and for now, we may assume that $\mathbf{\Delta}_{sr} = \mathbf{\Delta}_{rs}$. The matrix $\mathbf{\Delta}$ is often called a *Distance* or *Dissimilarity* matrix. In practice, we may construct a distance matrix from observations, but for the purposes of MDS we only require $\mathbf{\Delta}$.

The primary goal of MDS is to find a set of lower-dimensional points $\vec{y}_1, \ldots, \vec{y}_N \in \mathbb{R}^t$ corresponding to each of the N objects such that the distance between $\vec{y}_r$ and $\vec{y}_s$ is close to $\mathbf{\Delta}_{rs}$. There are numerous different notions of distance one

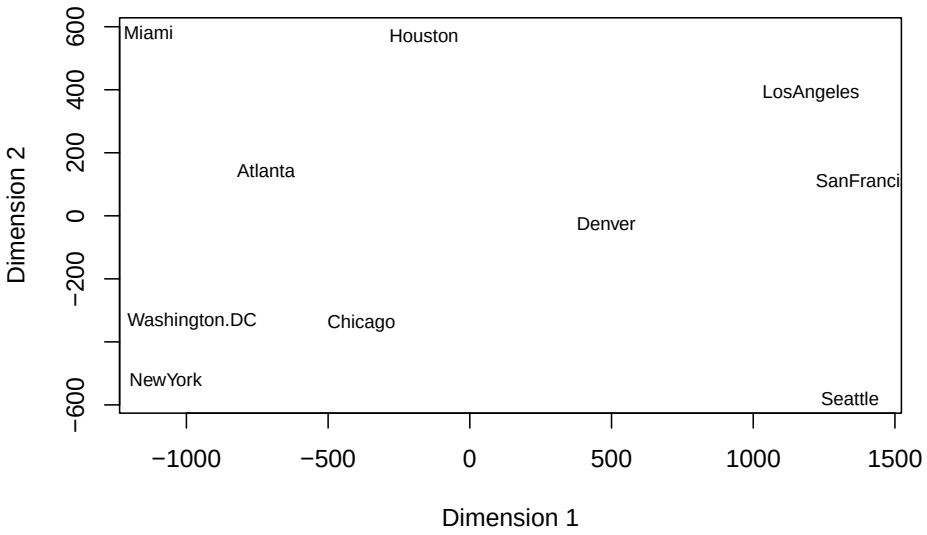
use in $\mathbb{R}^t$. Additionally, we may not know the notion of distance/dissimilarity used when computing $\mathbf{\Delta}$ or if the dissimilarity corresponds to any well defined notion of distance. Euclidean distance is a common choice for the distance of the lower-dimensional vectors. Fixing this choice still leaves many open questions. Are the original distances Euclidean? If so, can we determine the dimensionality of the original data?

MDS also serves as a data visualization method. In this case, t is typically chosen to be either two or three, and the N points in the two-dimensional (or three-dimensional) representation may be plotted so that one can visualize the relationships between data. Before turning to the most common methods of MDS, let's view a few examples using distances between cities.

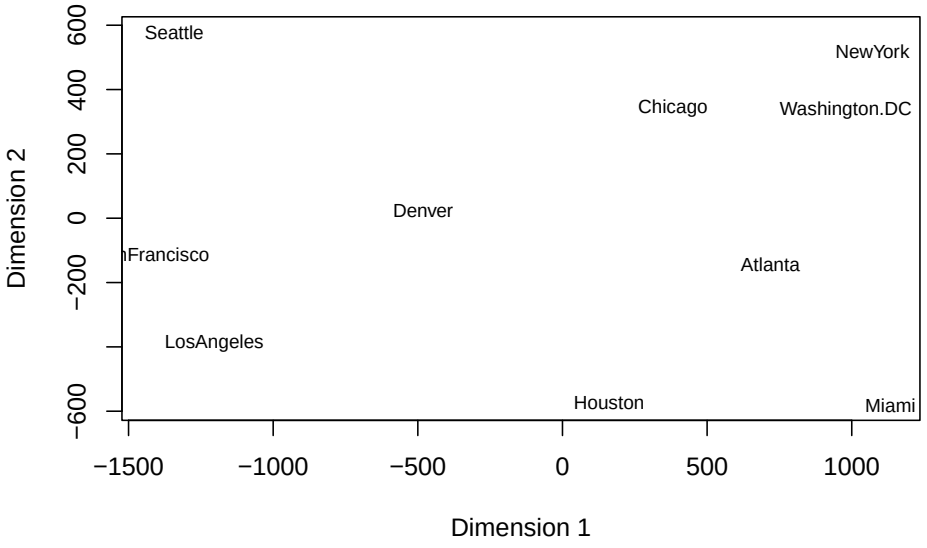
Example 4.3 (Classical scaling applied to distances between cities)**.** The *eurodist* R package provides air travel distances between 21 cities in Europe and 10 cities in the US. For the moment, we will focus on the 10 US cities with names and distances given in the following tables

	Atlanta	Chicago	Denver	Houston	LosAngeles	Miami	NewYork	SanFran
Atlanta	0	587	1212	701	1936	604	748	
Chicago	587	0	920	940	1745	1188	713	
Denver	1212	920	0	879	831	1726	1631	
Houston	701	940	879	0	1374	968	1420	
LosAngeles	1936	1745	831	1374	0	2339	2451	
Miami	604	1188	1726	968	2339	0	1092	
NewYork	748	713	1631	1420	2451	1092	0	
SanFrancisco	2139	1858	949	1645	347	2594	2571	
Seattle	2182	1737	1021	1891	959	2734	2408	
Washington.DC	543	597	1494	1220	2300	923	205	

After conducting classical scaling (a method of MDS, details will be discussed later), we acquire the plot below. The plot is consistent with the geographical relationships between the cities. Miami and Seattle are the farthest apart in our plot. NY and D.C. are quite close on the plot, which is also true for Los Angeles and San Francisco.



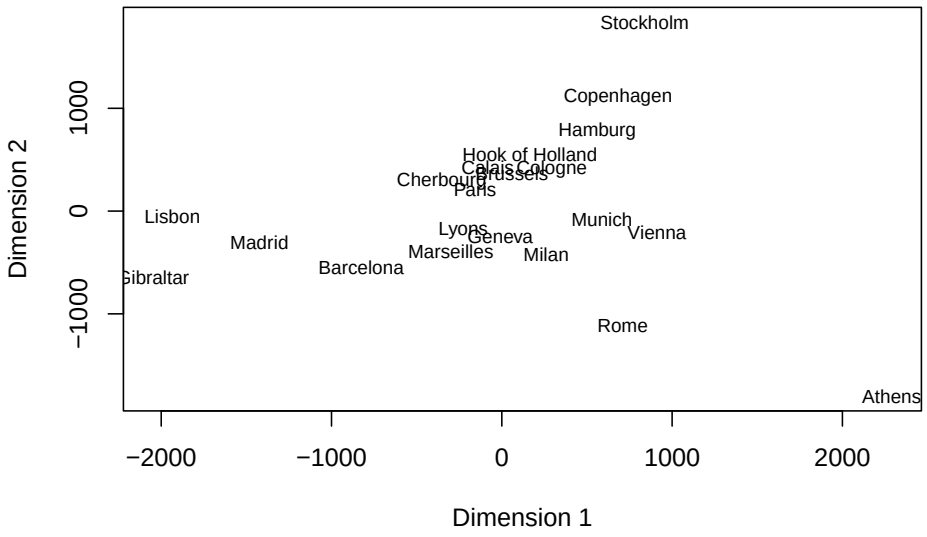
If we rotate the above plot 180 degrees, we recover an representation of the cities consistent with the typical map of the US.



Now let's try MDS on the 21 European cities

	Athens	Barcelona	Brussels	Calais	Cherbourg	Cologne	Copenhagen	Geneva	Gibra
Athens	0	3313	2963	3175	3339	2762	3276	2610	4
Barcelona	3313	0	1318	1326	1294	1498	2218	803	1
Brussels	2963	1318	0	204	583	206	966	677	2
Calais	3175	1326	204	0	460	409	1136	747	2
Cherbourg	3339	1294	583	460	0	785	1545	853	2

	Athens	Barcelona	Brussels	Calais	Cherbourg	Cologne	Copenhagen	Ge
Cologne	2762	1498	206	409	785	0	760	
Copenhagen	3276	2218	966	1136	1545	760	0	
Geneva	2610	803	677	747	853	1662	1418	
Gibraltar	4485	1172	2256	2224	2047	2436	3196	
Hamburg	2977	2018	597	714	1115	460	460	
Hook of Holland	3030	1490	172	330	731	269	269	
Lisbon	4532	1305	2084	2052	1827	2290	2971	
Lyons	2753	645	690	739	789	714	1458	
Madrid	3949	636	1558	1550	1347	1764	2498	
Marseilles	2865	521	1011	1059	1101	1035	1778	
Milan	2282	1014	925	1077	1209	911	1537	
Munich	2179	1365	747	977	1160	583	1104	
Paris	3000	1033	285	280	340	465	1176	
Rome	817	1460	1511	1662	1794	1497	2050	
Stockholm	3927	2868	1616	1786	2196	1403	650	
Vienna	1991	1802	1175	1381	1588	937	1455	



After applying a reflection of the map given by classical scaling to help it comply to the conventional orientation of a European map, the above plot reconstructs the European map quite well. Gibraltar, Lisbon and Madrid are in the south-west corner, the two North European cities Stockholm and Copenhagen are in the north end, and Athens is in the south-west corner.

Finally, let's consider the 18 representative global cities. Their pairwise flight lengths (geodesic distance) are shown in the table below. As we can see, the geodesic distances between the three Southern-Hemisphere cities: Rio, Cape

Town, Melbourne and other Northern-Hemisphere cities are generally large (almost all over 10,000 kilometers)

	Beijing	Cape Town	Hong Kong	Honolulu	London	Melbourne	Mexico City	Montreal	M
Beijing	0	12947	1972	8171	8160	9093	12478	10490	
Cape Town	12947	0	11867	18562	9635	10388	13703	12744	
Hong Kong	1972	11867	0	8945	9646	7392	14155	12462	
Honolulu	8171	18562	8945	0	11653	8862	6098	7915	
London	8160	9635	9646	11653	0	16902	8947	5240	
Melbourne	9093	10388	7392	8862	16902	0	13557	16730	
Mexico City	12478	13703	14155	6098	8947	13557	0	3728	
Montreal	10490	12744	12462	7915	5240	16730	3728	0	
Moscow	5809	10101	7158	11342	2506	14418	10740	7077	
New Delhi	2788	9284	3770	11930	6724	10192	14679	11286	
New York	11012	12551	12984	7996	5586	16671	3362	533	
Paris	8236	9307	9650	11988	341	16793	9213	5522	
Rio	17325	6075	17710	13343	9254	13227	7669	8175	
Rome	8144	8417	9300	12936	1434	15987	10260	6601	
S.F.	9524	16487	11121	3857	8640	12644	3038	4092	
Singapore	4465	9671	2575	10824	10860	6050	16623	14816	
Stockholm	6725	10334	8243	11059	1436	15593	9603	5900	
Tokyo	2104	14737	2893	6208	9585	8159	11319	10409	

The three-dimensional visualization result of classical MDS is shown above. You

can rotate and magnify it on your laptop. The blue points represent Asian cities, the black points represent European cities, and the red points represent North American cities. If you inspect the plot clearly, you may notice that the cities appear to be constrained to the surface of a sphere, which complies to the true scenario.

In each of the examples, classical MDS was quite successful in generating maps which reflected the geographical configuration with continental or global maps which were (after some reflections/rotations) consistent with conventional maps. Let's discuss the details.

4.4.1 Key features of MDS

MDS can be divided into three major types: Classical Scaling; Metric MDS; and Non-Metric MDS. The choice of method depends on the specifics of the distance/dissimilarity matrix and features which are preferential to preserve. Classical Scaling and Metric MDS require that the provided distances correspond to a metric (more on this below), while Non-metric MDS is usually used when the input data doesn't satisfy the properties of a true distance metric or when the relationships are ordinal (i.e., we only know which distances are larger, but not by how much).

Beyond demonstrating the capacity of MDS to recover meaningful visualization, we also gain two insights from the examples above which hold for any MDS algorithm.

i) As we can tell from the recovery of US map and European map, the configurations of $\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_N$ are not unique, as we can rotate or flip the map. Actually, if $\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_N \in \mathbb{R}^{t'}$ is considered as the optimal solution using Euclidean distance, then given any vector $\vec{b} \in \mathbb{R}^{t'}$ and orthogonal matrix $A \in \mathbb{R}^{t' \times t'}$, $||(A\tilde{y}_r + \vec{b} - (A\tilde{y}_s + \vec{b})|| = ||\tilde{y}_r - \tilde{y}_s||$. Rotation, Reflection or Translation don't alter the pairwise distances. So $A\tilde{y}_1 + \vec{b}, A\tilde{y}_2 + \vec{b}, \ldots, A\tilde{y}_N + \vec{b}$ is also an optimal solution. Outside of Euclidean distances, reflections are still possible which eliminate the possibility of unique configurations.

ii) The pairwise distance between two objects need not be Euclidean. In the above example, they are actually great circle distance.

In the following subsections, we will discuss the specifics of classical scaling with a brief discussion on two common methods of MDS, one each in metric and nonmetric MDS.

4.4.2 Classical Scaling

Let us first introduce some important definitions.

Definition 4.3 (Distance Matrix)**.** A matrix $\Delta \in \mathbb{R}^{N \times N}$ is called a distance matrix if it possesses the following properties:

a) Symmetry: Δ is symmetric, meaning that: $\Delta_{rs} = \Delta_{sr} \quad$ for all r and s

b) Zero Diagonal: The diagonal entries of the matrix represent the distance of a point to itself, and are thus all zeros: $\Delta_{rr} \equiv 0 \quad$ for all $1 \leq r \leq N$

c) Non-negativity: All distances are non-negative: $\Delta_{rs} \geq 0$

d) Triangle Inequality: The distances in the matrix respect the triangle inequality:

$$\Delta_{rs} \leq \Delta_{rt} + \Delta_{ts}$$

Distance matrices can be computed in many different ways. We will focus on the following specific case.

Definition 4.4 (Euclidean Distance Matrix)**.** A distance matrix $\Delta \in \mathbb{R}^{N \times N}$ is a Euclidean distance matrix if there exists a configuration $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N$ s.t. Δ_{rs} represents the Euclidean distance between points r and s, i.e., $||\vec{y}_r - \vec{y}_s|| = \Delta_{rs} \forall r, s$.

Classical scaling operates under the assumption that Δ is a Euclidean distance matrix though the dimensionality of the configuration $\vec{y}_1, \ldots, \vec{y}_N$ which gives rise to the distances specified by Δ is unknown. As we shall see, the minimum dimensionality needed for the configuration to give the specified distances (assuming it exists at all) will be discovered through the classical scaling algorithm. Furthermore, since we are free to translate any configuration without altering pairwise distance, we will seek a configuration $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N$ which is centered to simplify the following computation. The rotation of the configuration will also be given so that an optimal solution in $k-1$ dimensions can be attained by dropping the final coordinate in the k dimensional configuration.

4.4.2.1 Recovering Coordinates

After some preprocessing, finding a centered configuration $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N$ from the Euclidean distance matrix Δ is possible through some of the linear algebra techniques we have discussed so far. Observe that Δ_{rs}^2 can be expressed as

$$\Delta_{rs}^2 = ||\vec{y}_r||^2 + ||\vec{y}_s||^2 - 2\vec{y}_r^T \vec{y}_s$$

so that

$$\vec{y}_r^T \vec{y}_s = -\frac{1}{2}\left(\Delta_{rs} - \|\vec{y}_r\|^2 - \|\vec{y}_s\|^2\right).$$

We will use this relationship to build an inner product matrix $\mathbf{B}$ with entries $\mathbf{B}_{rs} = \vec{y}_r^T \vec{y}_s$. In matrix form, we may write

$$\mathbf{B} = \mathbf{Y}\mathbf{Y}^T$$

, where $\mathbf{Y}$ is a data matrix containing the configuration $\vec{y}_1, \ldots, \vec{y}_N$ with Euclidean distances Δ.

1) Computing the inner product matrix B

The inner product term can be represented as: $B_{ij} = \vec{y}_i^T \vec{y}_j$. As a result,

$$(\Delta_{ij})^2 = B_{ii} + B_{jj} - 2B_{ij}.$$

Key Observation In the previous line, we express the entries of Δ (known) with entries of B (unknown). Our goal is to find a way to express entries of B (unknown) with entries of Δ. Intuitively, we want to cancel out B_{ij} terms in the expression. Considering that $\sum_{i=1}^{N} \vec{y}_i = \vec{0}$, we sum both sides of the equation over the index i. We can get the following expression:

$$\sum_{i=1}^{N} (\Delta_{ij})^2 = \mathrm{tr}(B) + NB_{ii}$$

Similarly, we can also sum both sides of the equation over index j, and get the expression:

$$\sum_{j=1}^{N} (\Delta_{ij})^2 = \mathrm{tr}(B) + NB_{jj}$$

We successfully eliminate all the off-diagonal terms of B through the above steps. Now, we want to take a step further. Sum both sides of the equations over both indexes i and j. We acquire the following expression:

$$\sum_{i=1}^{N} \sum_{j=1}^{N} (\Delta_{ij})^2 = 2N \,\mathrm{tr}(B)$$

Now we can solve the entries of Δ using the entries of B through a backward calculation. From the last equation, we get

$$\mathrm{tr}(B) = \frac{1}{2N} \sum_{i=1}^{N} \sum_{j=1}^{N} \Delta_{ij}^2$$

Then substitute the above expression into above formulas, we get the expression of the diagonal entries of B:

$$B_{ii} = \frac{1}{N} (\sum_{j=1}^{N} (\Delta_{ij})^2 - \mathrm{tr}(B))$$

After that, we can finally get the off-diagonal entries of B:

$$\begin{aligned} B_{ij} &= \frac{1}{2}(B_{ii} + B_{jj} - (\Delta_{ij})^2) \\ &= -\frac{1}{2}(\Delta_{ij})^2 + \frac{1}{N} \sum_{i=1}^{N} (\Delta_{ij})^2 + \frac{1}{N} \sum_{j=1}^{N} (\Delta_{ij})^2 - \frac{1}{2N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} (\Delta_{ij})^2 \end{aligned}$$

We may more compactly express the inner product matrix B in a matrix form, as

$$B = HAH$$

where $A \in \mathbb{R}^{N \times N}$ has entries $A_{ij} = -\frac{1}{2}(\Delta_{ij})$ for $\forall\ 1 \leq i, j \leq N$ and H is the centering matrix

$$H = \mathbb{I}_N - \frac{1}{N}\mathbb{1}_N\mathbb{1}_N^T$$

.

2) Recover the coordinates using inner product matrix B

Both diagonal and off-diagonal entries of the inner product matrix B has been shown. We assumed that $B = YY^T$, so B is symmetric and positive semi-definite (all eigenvalues are non-negative). Assuming B is rank t implies that it has t positive eigenvalues and $N - t$ 'zero' eigenvalues.

Our intuition is to apply SVD/diagonalization to B in order to recover the configuration $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N \in \mathbb{R}^t$ by computing the nonzero eigenvalues and corresponding eigenvectors giving the factorization.

$$\begin{aligned} B &= (\vec{u}_1 \,|\vec{u}_2| \ldots |\, \vec{u}_t) \begin{pmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_t \end{pmatrix} \begin{pmatrix} \vec{u}_1^\top \\ \vec{u}_2^\top \\ \vdots \\ \vec{u}_t^\top \end{pmatrix} \\ &= \tilde{U} \begin{pmatrix} \lambda_1^{1/2} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_t^{1/2} \end{pmatrix} \begin{pmatrix} \lambda_1^{1/2} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_t^{1/2} \end{pmatrix} \tilde{U}^T \\ &= (\tilde{U}\Lambda^{1/2})(\tilde{U}\Lambda^{1/2})^T \end{aligned}$$

Let $Y = \tilde{U}\Lambda^{1/2}$, so that rows of $\tilde{U}\Lambda^{1/2}$ correspond to the vectors $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N$. It satisfies every entry of the inner product matrix B, as well as all pairwise distances Δ_{ij}! Importantly, the rank of B immediately indicates the minimal dimension needed to recover a configuration which exactly recovers the Euclidean distances in Δ.

For visualization, we may prefer a lower dimensional configuration than the dimensionality t discovered through classical scaling. For a $k < t$ dimensional configuration, we instead use the first k columns of $\tilde{U}\Lambda^{1/2}$ as a our k-dimensional vectors. Outside of optimal recovery, we may instead formulate the classical scaling problem in k-dimensions as the following minimization problem.

$$\text{argmin}_{Y \in \mathbb{R}^{N \times k}} \|B - YY^T\|_F$$

from which it follows that YY^T is built from the rank k approximation to B. Using the results from SVD and the equivalence of SVD and eigendecompositions for symmetric positive, semidefinite matrices, this is equivalent to using the first k columns of $\tilde{U}\Lambda^{1/2}$.

Dealing with real data case

For a given distance matrix, the Euclidean condition may not hold. In the previous globe map example, the distance matrix is based on the geodesic distance instead of the Euclidean distance. Under this circumstance, the "inner product" matrix $B = HAH$ is still symmetric but will not be positive definite. As a simpler example, consider the following case

Example 4.4 (Non Euclidean Distances)**.** Suppose we have a distance matrix Δ:

Given the matrix:

$$\Delta = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 0 \end{pmatrix}$$

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    1    1    2
## [2,]    1    1    0    1
## [3,]    1    1    0    1
## [4,]    2    1    1    0
```

i) Compute matrix A as:

$$A = -\frac{1}{2}\Delta^2$$

```
##      [,1] [,2] [,3] [,4]
## [1,]  0.0 -0.5 -0.5 -2.0
## [2,] -0.5 -0.5  0.0 -0.5
## [3,] -0.5 -0.5  0.0 -0.5
## [4,] -2.0 -0.5 -0.5  0.0
```

ii) Compute matrix B using:

$$H = I - \frac{1}{n}\mathbf{1}\mathbf{1}^T$$

Where I is the identity matrix and n is the number of rows (or columns) in Δ. Then:

$$B = HAH$$

```
##         [,1]    [,2]    [,3]    [,4]
## [1,]  0.9375  0.1875 -0.0625 -1.0625
## [2,]  0.0625 -0.1875  0.0625  0.0625
## [3,]  0.0625 -0.1875  0.0625  0.0625
## [4,] -1.0625  0.1875 -0.0625  0.9375
```

iii) Finally, perform an eigen-decomposition on matrix B which gives eigenvalues.

```
## [1]  2.00 -0.25  0.00  0.00
```

Here, we have a negative eigenvalue $-\frac{1}{4}$. This indicates the original distance matrix Δ is not a Euclidean distance matrix; there is no Euclidean space containing four vectors with the given pairwise distances!!

Other practical issues

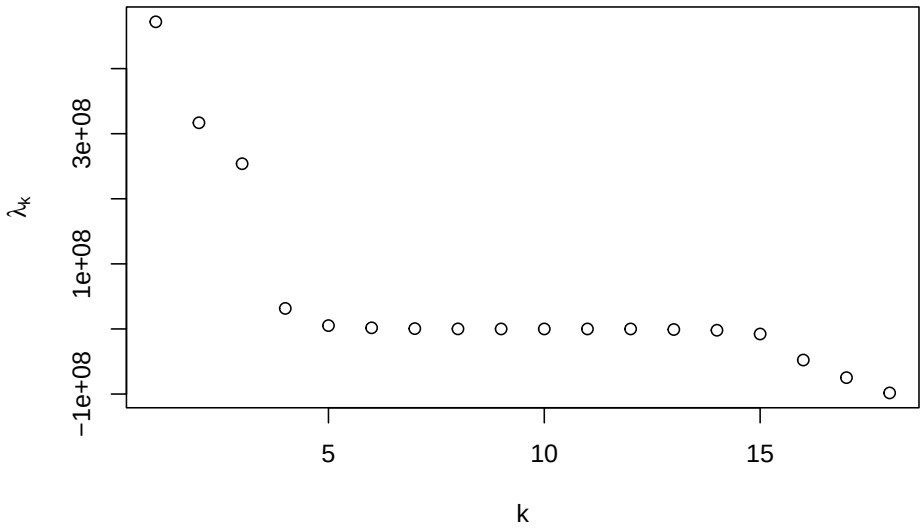
1) **Asymmetric Δ:** There are cases where the distance matrix Δ is not symmetric. In this case, we usually set $\Delta \leftarrow \frac{1}{2}(\Delta + \Delta^T)$ to enforce symmetry.

2) When there exist some negative eigenvalues in the inner product matrix B, we usually have two options to deal with it.

   a) Inflate the original proximity matrix Δ by a small constant factor c, i.e., $\Delta_{ij} \leftarrow \Delta_{ij} + c$, if $i \neq j$ until B is positive semidefinite.

   b) If there exist several negative eigenvalues with small absolute value (compared to the largest several positive eigenvalues), and there are more positive eigenvalues than our prior estimation (the dimension of the original configuration), we may just pick the largest t eigenvalues and eliminate the rest.

Example 4.5 (Global Cities Revisited)**.** We can also consider the previous global city distance matrix example. Plot the scree plot of the inner product matrix.



We find that the first three eigenvalues are much larger than the rest, so we assume that the dimension of the original configuration is 3, which also complies

to our knowledge about global map.

4.4.2.2 Duality of PCA and Classical Scaling

"There is a duality between a principals components analysis and classical MDS where dissimilarities are given by Euclidean distance"[**?**].

Given the matrix expression of the original configuration $\mathbf{X} \in \mathbb{R}^{N \times d}$. Recall that PCA is attained by finding the eigen-vectors of the covariance matrix $\frac{1}{N-1}(HX)^T(HX)$, where H is the centering matrix. Suppose the k eigen-vectors are $\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_k$, and the corresponding eigenvalues are $\mu_1, \mu_2, \ldots, \mu_k$. Then $HX = YW^T$, where $W = (\vec{w}_1 | \vec{w}_2 | \ldots | \vec{w}_N)$ represents PC Loadings, and Y represents PC Scores.

While MDS is attained by first converting X into distance matrix, here, Euclidean distance. In the classical MDS algorithm, the process of converting the Euclidean distance matrix into the inner product matrix gives $B = (HX)(HX)^T$. then we find the eigenvectors of B which are equivalent to the left singular vectors of HX and with nonzero eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_d$ which are the squared singular values of HX.

Recall what you have learned in Linear Algebra. The eigenvalues of $HX(HX)^T$ are the same as those for X^TX, together with an extra n-p zero eigenvalues. So the first t PC scores give the t-dimensional configuration for Classical Scaling. **Choosing the number of dimensionality t of the original configuration is equivalent to choosing the number of principal components to keep.**

Furthermore, we have shown that $HX = YW^T$, then $B = (HX)(HX)^T = YW^TWY^T = YY^T$, this means **PC scores are the exact solutions for Classical Scaling!** As a result, classical MDS has the same strengths and weaknesses of PCA.

4.4.3 Metric MDS

Metric MDS is a set of algorithms which seek optimal configuration which minimize a specified loss function. Typically, this loss function is called the stress which has the general form

$$S(\vec{y}_1, \ldots, \vec{y}_N) = \sum_{i<j} W_{ij} \left(\|\vec{y}_i - \vec{y}_j\|^2 - \Delta_{ij} \right)^2 = \frac{1}{2} \sum_{i \neq j} W_{ij} \left(\|\vec{y}_i - \vec{y}_j\|^2 - \Delta_{ij} \right)^2$$

where the weights $W_{ij} \geq 0$ have different conventions which prioritize the preservation of certain distances in the original coordinates. Heuristically, you can think of $\left(\|\vec{y}_i - \vec{y}_j\|^2 - \Delta_{ij} \right)^2$ as measuring how far the lower-dimensional Euclidean distances $\|\vec{y}_i - \vec{y}_j\|$ deviate from the specified original distance Δ_{ij}. Importantly, there is no closed form expression for $\vec{y}_1, \ldots, \vec{y}_N$ which minimize S so gradient based optimization methods are typically used in practice. When the desired dimension is unknown, one can estimate different $\vec{y}_1, \ldots, \vec{y}_N$ in different

dimensions and compare the optimal stress as a data driven method for choosing the dimensionality.

A important method of metric MDS is the Sammon mapping which takes $W_{ij} \propto \frac{1}{\Delta_{ij}}$. Thus, if $\Delta_{ij} \gg \Delta_{k\ell}$ it follows that $W_{ij} \ll W_{k\ell}$ and we may infer that the Sammon mapping places a greater emphasis on the preservation of small distances! Briefly, we show the results of the Sammon mapping applied to the global cities data.

Example 4.6 (Sammon mapping and global cities)**.** First, we show the minimal Sammon stress at each dimension.

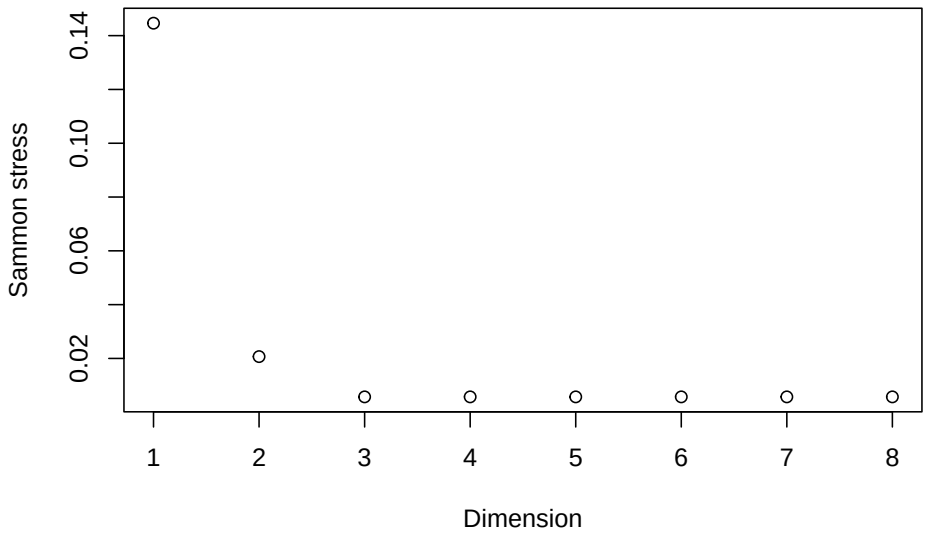


Figure 4.6: Sammon stress for Global Cities Data

```
## A marker object has been specified, but markers is not in the mode
## Adding markers to the mode...
```

This stress saturates near a small value for dimensions three and up suggesting a three dimensional configuration is suitable. We show this result below. Note its similarities to the result of classical scaling (which is default initialization for the Sammon mapping optimization algorithm).

4.4.4 Nonmetric MDS

To be added in the next edition.

4.5 Exercises

1. Show that the PCA scores are centered.
2. Suppose a data matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$ has principal component scores $\mathbf{Y} \in \mathbb{R}^{N \times d}$, principal component loading matrix $\mathbf{W} \in \mathbb{R}^{d \times d}$, and principal component variances $\lambda_1, \ldots, \lambda_d$. Show that the sample covariance of the PCA scores, $\mathbf{Y}$, is diagonal.
3. Consider the data matrix

$$\mathbf{X} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix}.$$

   a. Compute the principal component scores, variances, and loadings of $\mathbf{X}$.

b. Does $\mathbf{X}$ exhibit lower dimensional structure? If so, describe it.

4. Given a dataset with outliers:

$$\mathbf{X} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 3 & 6 \\ 4 & 8 \\ 100 & 200 \end{bmatrix}$$

   a. Center the data.
   b. Compute the covariance matrix.
   c. Perform PCA and identify the principal component scores, loadings, and variances.
   d. Discuss how outliers affect PCA and suggest ways to handle them.

5. Consider a dataset:

$$\mathbf{X} = \begin{bmatrix} 1 & 100 \\ 2 & 200 \\ 3 & 300 \\ 4 & 400 \end{bmatrix}$$

   a. Center the data.
   b. Compute the covariance matrix.
   c. Perform PCA and find the principal components.
   d. Discuss the importance of feature scaling in PCA.

6. Given a nonlinear dataset:

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 2 & 4 \\ 3 & 9 \\ 4 & 16 \\ 5 & 25 \end{bmatrix}$$

   a. Center the data.
   b. Compute the covariance matrix.
   c. Perform PCA and identify the principal components.
   d. Discuss the limitations of PCA for nonlinear datasets and suggest alternative methods.

7. Load the Iris dataset and standardize the features.

   a. Perform PCA on the standardized data.
   b. Plot the cumulative explained variance as a function of the number of principal components.
   c. Choose the number of principal components that explain at least 95% of the variance.
   d. Project the data onto the chosen principal components and visualize the results.

e. Discuss the results and the effectiveness of PCA in reducing the dimensionality of the dataset.

8. Suppose
$$\vec{x} = (x_1, x_2)^T \sim \mathcal{N}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1 & \rho\\ \rho & 1\end{bmatrix}\right),$$
i.e. random vector $\vec{x}$ follows a multivariate normal distribution with mean $\vec{0}$ and covariance $\Sigma_X = \begin{bmatrix}1 & \rho\\ \rho & 1\end{bmatrix}$. Let $w = \frac{\sqrt{2}}{2}(x_1 + x_2)$ and $z = \frac{\sqrt{2}}{2}(x_1 - x_2)$. Find the joint distribution of $(w, z)^T$. Hint: linear combinations of normal random variables are also normal.

9. Consider a data matrix $X \in \mathbb{R}^{N\times d}$ with centered columns so that the sample covariance matrix is
$$\hat{\Sigma} = \frac{X^T X}{N}.$$
Assume $\hat{\Sigma}$ has eigenvalues $\lambda_1 > \lambda_2 > \cdots > \lambda_d > 0$ with orthonormal eigenvectors $\vec{w}_1, \ldots, \vec{w}_d$.

   a. If $X^{(1)} = X - X\vec{w}_1\vec{w}_1^T$ is the data matrix where each row has had its component in the direction of $\vec{w}_1$ removed, show that
$$\hat{\Sigma}^{(1)} = \frac{{X^{(1)}}^T X^{(1)}}{N} = \hat{\Sigma} - \lambda_1\vec{w}_1\vec{w}_1^T.$$

   b. Show that $\hat{\Sigma}$ can be written in the form $\hat{\Sigma} = \sum_{j=1}^{d}\lambda_j\vec{w}_j\vec{w}_j^T$.

10. For $k < d$, let $\vec{q}_1, \ldots, \vec{q}_k \in \mathbb{R}^d$ be fixed orthonormal vectors. Suppose $a_1, \ldots, a_k$ are independent Gaussian random variables with mean zero and variances $\lambda_1 > \cdots > \lambda_k$ respectively. Let
$$\vec{x} = a_1\vec{q}_1 + \cdots + a_k + \vec{q}_k + \vec{\epsilon}$$
where $\vec{\epsilon} \sim \mathcal{N}(\vec{0}, \sigma^2\mathbf{I})$ is independent of $a_1, \ldots, a_k$

   a. Find the mean and covariance of $\mathbf{X}$.
   b. Find the eigenvalues and eigenvectors of $\mathbf{\Sigma}$.
   c. For $d = 10$ and \$k=3 and let $\lambda_1 = 25$, $\lambda_2 = 9$, $\lambda_3 = 4$ and $\sigma^2 = 1$. Generate 100 independent realizations of $\vec{x}$ and compute the principal component loadings and variances. How do these compare to your results from b.
   d. Repeat c. for 10^4 samples. How have the results changed?

11. Let $\vec{x}_1, \ldots, \vec{x}_N$ be vectors in $\mathbb{R}^d$. Assume a PCA of these data has loadings $\vec{w}_1, \ldots, \vec{w}_d$ with associated variances $\lambda_1 \geq \cdots \geq \lambda_d \geq 0$. Let U be a $d \times d$ orthonormal matrix and set $\vec{y}_i = U\vec{x}_i$. Find expressions for the principal

component loadings and variance of $\vec{y}_1, \ldots, \vec{y}_N$ in terms of U, $\vec{w}_1, \ldots, \vec{w}_d$ and $\lambda_1, \ldots, \lambda_d$.

12. Load the *mtcar* dataset containing 11 observations from 32 cars.

    a. Show the principal component loadings and a biplot of the first and second PC components.

    b. Rescale the mpg (miles per gallon) data to feet per gallon, i.e. *mtcars$mpg <- 5280*mtcars$mpg.* Rerun PCA on these modified data and show the loadings and biplot.

    c. What is the first loading capturing? Explain this result.

    d. Rerun the results using the empirical correlation matrix by setting the option *scale = TRUE* in the prcomp command. Compare this result with part (a)

13. Consider the helix data shown below from two different directions.

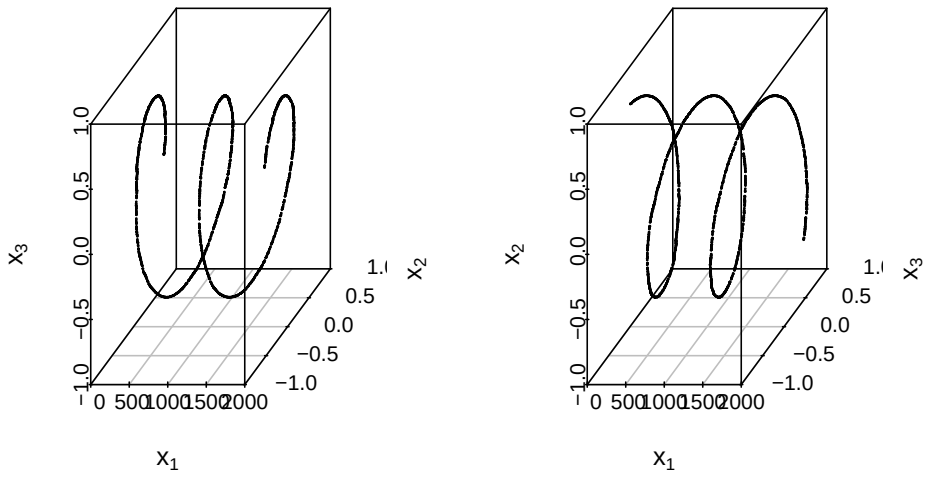


Figure 4.7: Helix data

a. What is the dimension of the shape formed by these data?

b. Compute the three principal component variances and show them below.

c. Do the principal component variances reflect the dimension of the data? Why or why not?

1. Consider a dataset with three features:

$$\mathbf{X} = \begin{bmatrix} 2 & 0 & 1 \\ 3 & 2 & 2 \\ 4 & 4 & 3 \\ 5 & 6 & 4 \\ 6 & 8 & 5 \end{bmatrix}$$

a. Center the data by subtracting the mean of each feature.
b. Compute the covariance matrix of the centered data.
c. Find the eigenvalues and eigenvectors of the covariance matrix.
d. Project the data onto the first two principal components.
e. Visualize the original data and the projected data in a 2D plot.
f. Discuss how PCA can help in reducing the dimensionality of the data while preserving important information.

2. The dataset *simplex10* contains samples generated from the probability simplex in $\mathbb{R}^{10}$. (This means all entries nonnegative and each random vector has entries which sum to one.)

   a. Run PCA, SVD, and NMF on these data and compare the results. In particular, what lower dimensional structure, if any, do these methods indicate?

   b. Why do the results differ?

3. Show that the Frobenius norm of a matrix is the square of the sum of its squared singular values. You may use the helpful identity $\|\mathbf{A}\|_F^2 = tr(\mathbf{A}\mathbf{A}^T)$

Chapter 5

Kernels and Nonlinearity

The techniques considered in the previous chapter (PCA, NMF, SVD, and classical Scaling) are ill suited to identify nonlinear structure and dependence in data. If we wish to most efficiently reduce dimensions without loss of information, we will need techniques which incorporate nonlinear structure. One can expand a data matrix by including specific nonlinear relationships then apply PCA or SVD but there are numerous problems with this approach. In particular, which relationships does one choose to include? Even including simple quadratic or cubic terms (features) can result in a data matrix with a massive increase in the number of columns. Even when the original dimensionality of the data is moderate, the including of polynomial terms can quickly result in a data matrix of nonlinear features with an untenable number of columns which can make application of the linear methods we have discussed much more computationally demanding to implement.

Kernels are a important class of functions which can be used to `kernelize` the methods we have discussed before. In theory, these kernelized versions of the linear methods we have discussed can identify and use nonlinear structure for better dimensionality reduction while circumventing the issue of higher dimensional `featurized` data. This approach follows from an application of the so called 'kernel trick' which we now discuss.

Briefly, a kernel is a function

$$k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$$

which has an associated feature space, $\mathcal{H}$ and (implicity defined, possibly nonlinear) feature mapping $\varphi : \mathcal{R}^d \to \mathcal{H}$ such that inner products in the feature space, denoted $\langle \varphi(\vec{x}), \varphi(\vec{y}) \rangle_{\mathcal{H}}$ can be obtained through an evaluation of the kernel, namely

$$k(\vec{x}, \vec{y}) = \langle \varphi(\vec{x}), \varphi(\vec{y}) \rangle_{\mathcal{H}} \tag{5.1}$$

Any method which can be expressed involving inner products can be kernelized by replacing terms of the form $\vec{x}_i^T \vec{x}_j$ with the quantity $k(\vec{x}_i, \vec{x}_j)$. Thus, we are replacing inner products of our original d-dimensional data with inner products in the associated feature space $\mathcal{H}$. Importantly, if we only need inner products, we never need to explicitly compute the feature map φ for any of our data! At first glance this connection may seem minor, but by using kernels we can turn many linear techniques into nonlinear methods including PCA, SVD, support vector machines, linear regression, and many others.

There are some limits though. Not every choice of k has an associated feature space. A function is only a kernel if it satisfies Mercer's Condition.

Theorem 5.1 (Mercer's Condition)**.** *A function*

$$k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$$

has a an associated feature space $\mathcal{H}$ and feature mapping $\varphi : \mathbb{R}^d \to \mathcal{H}$ such that

$$k(\vec{x}, \vec{y}) = \langle \varphi(\vec{x}), \varphi(\vec{y}) \rangle_{\mathcal{H}}, \qquad \forall \vec{x}, \vec{y} \in \mathbb{R}^d$$

if and only if for any $N \in \{1, 2, \dots\}$ and $\vec{x}_1, \dots, \vec{x}_N \in \mathbb{R}^d$ the kernel matrix $\mathbf{K} \in \mathbb{R}^N$ with entries $\mathbf{K}_{ij} = k(\vec{x}_i, \vec{x}_j)$ is positive semidefinite. Equivalently, it must be the case that

$$\int_{\mathbb{R}^d} \int_{\mathbb{R}^d} g(\vec{x}) g(\vec{y}) k(\vec{x}, \vec{y}) d\vec{x} d\vec{y} \geq 0$$

whenever $\int_{\mathbb{R}^2} [g(\vec{x})] d\vec{x} < \infty$.

We will only consider symmetric functions such that $k(\vec{x}, \vec{y}) = k(\vec{y}, \vec{x})$ for all $\vec{x}, \vec{y} \in \mathbb{R}^d$. It may not be immediately obvious if a symmetric function satisfies Mercer's condition, but there are many known examples. A few are shown in the following table.

Name	Equation	Tuning Parameters
Radial Basis Function	$k(\vec{x}, \vec{y} = \exp\left(-\sigma \|\vec{x} - \vec{y}\|^2\right)$	Scale $\sigma > 0$
Laplace	$k(\vec{x}, \vec{y} = \exp\left(-\sigma \|\vec{x} - \vec{y}\|\right)$	Scale $\sigma > 0$
Polynomial	$k(\vec{x}, \vec{y}) = (c + \vec{x}^T \vec{y})^d$	Offset $c > 0$, Degree $d \in \mathbb{N}$

The radial basis function (rbf) is the most commonly used kernel and has an associated feature space $\mathcal{H}$ which is infinite dimensional! The associated feature map φ for the rbf kernel is

$$\varphi(\vec{x}) = e^{-\sigma \|\vec{x}\|^2} \left(a_{\ell_0}^{(0)}, a_1^{(1)}, \dots, a_{\ell_1}^{(1)}, a_1^{(2)}, \dots, a_{\ell_2}^{(2)}, \dots \right)$$

where $\ell_j = \binom{d+j-1}{j}$ and $a_\ell^{(j)} = \frac{(2\sigma)^{j/2} x_1^{\eta_1} \ldots x^{\eta_d}}{\sqrt{\eta_1! \ldots \eta_d!}}$ when $\eta_1 + \cdots + \eta_d = j$. The preceding expression is quite cumbersome, but there is one important point to emphasize. Every possible polynomial combination of the coordinates of $\vec{x}$ appears in some coordinate of $\varphi(\vec{x})$ (though higher order terms are shrunk by the factorial factors in the denominator of $a_\ell^{(j)}$). Thus, the rbf kernel is associated with a very expressive feature space which makes it a potent but dangerous choice since risks overfitting. To explore these details more, let's discuss one very important application of kernels in unsupervised learning.

5.1 Kernel PCA

Suppose we have a kernel k and associated feature map φ. In kernel PCA, we want to apply to PCA to the featurized data $\varphi(\vec{x}_1), \ldots, \varphi(\vec{x}_N)$ rather than the original data. The idea is that by studying the featurized data, we can identify additional nonlinear structure in the features that provides a better lower-dimensional representation of the data. We have discussed three approaches to computing PC scores to data: (i) diagonalization of the sample covariance, (ii) applying SVD to the centered data, and (iii) using the duality of PCA and classical scaling.

For the rbf kernel and its infinite dimensional feature map, approaches (i) and (ii) are impossible. Why? The centered data matrix of features

$$\mathbf{H}\tilde{\mathbf{X}} = \mathbf{H} \begin{bmatrix} \varphi(\vec{x}_1)^T \\ \vdots \\ \varphi(\vec{x}_N)^T \end{bmatrix} = \begin{bmatrix} \varphi(\vec{x}_1)^T - \bar{\varphi}^T \\ \vdots \\ \varphi(\vec{x}_N)^T - \bar{\varphi}^T \end{bmatrix}$$

has a infinite number of columns so that we cannot compute its SVD. In the above expression, $\bar{\varphi} = \frac{1}{N}\sum_{i=1}^N \varphi(\vec{x}_i)$ is the mean for the feature vectors. The associated sample covariance matrix

$$\mathbf{\Sigma}_F = \frac{1}{N}\tilde{\mathbf{X}}\mathbf{H}\tilde{\mathbf{X}} = \frac{1}{N}\sum_{i=1}^N \left(\varphi(\vec{x}_i) - \bar{\varphi}\right)\left(\varphi(\vec{x}_i) - \bar{\varphi}\right)^T$$

will have an infinite number of rows and columns so we cannot hope to diagonalize it either.

Fortunately, the third option, using duality of classical scaling and PC, provides a workaround. Observe that the inner product matrix of the centered feature data $\mathbf{H}\tilde{\mathbf{X}}(\mathbf{H}\tilde{\mathbf{X}})^T$ can be written in terms of the kernel since

$$\mathbf{H}\tilde{\mathbf{X}}(\mathbf{H}\tilde{\mathbf{X}})^T = \mathbf{H} \begin{bmatrix} \varphi(\vec{x}_1)^T \\ \vdots \\ \varphi(\vec{x}_N)^T \end{bmatrix} \begin{bmatrix} \varphi(\vec{x}_1) & \ldots & \varphi(\vec{x}_N) \end{bmatrix} \mathbf{H} = \mathbf{HKH}$$

where $\mathbf{K}$ has the inner products in the feature space which we can calculate using the kernel function

$$\mathbf{K}_{ij} = \varphi(\vec{x}_i)^T \varphi(\vec{x}_j) = k(\vec{x}_i, \vec{x}_j).$$

Since k is a symmetric kernel, it follows that $\mathbf{K}$ is positive semidefinite. Using this property, one can argue that $\mathbf{HKH}$ will also be positive semidefinite. We can use the eigendecomposition of the doubly centered kernel to compute the kernel principal component scores. Specifically, if $\mathbf{HKH}$ rank r with eigenvalues $\lambda_1 \geq \cdots \geq \lambda_r > 0$ and corresponding eigenvalues $\vec{u}_1, \ldots, \vec{u}_r \in \mathbb{R}^N$, then $\mathbf{HKH}$ factorizes as

$$\mathbf{HKH} = \underbrace{\begin{bmatrix} \vec{u}_1 & \ldots & \vec{u}_r \end{bmatrix} \begin{bmatrix} \lambda_1^{1/2} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_r^{1/2} \end{bmatrix}}_{\mathbf{U}\mathbf{\Lambda}^{1/2}} \left(\mathbf{U}\mathbf{\Lambda}^{1/2}\right)^T.$$

The rows of the matrix $\mathbf{U}\mathbf{\Lambda}^{1/2}$ are almost the kernel PC scores. The only issue is an additional the identity

$$\mathbf{HKH} = (\mathbf{H}\tilde{\mathbf{X}})(\mathbf{H}\tilde{\mathbf{X}})^T$$

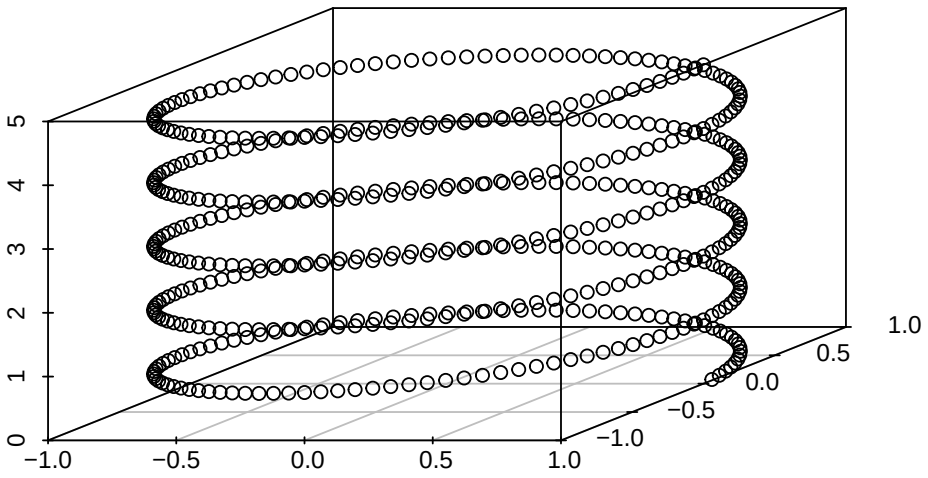
is missing the factor of $1/N$ appearing in the covariance calculation. Accounting for this, the first r non-zero kernel PC scores are the rows of the matrix

$$\frac{1}{\sqrt{N}}\mathbf{U}\mathbf{\Lambda}^{1/2}$$

and the corresponding nonzero PC variances are $\lambda_1/N, \ldots, \lambda_r/N$.

Notably, at no point do we compute the PC loadings! However, similar to standard PCA, we use the scores for dimension reduction and the PC variances for choosing a dimension. Without the loadings, we cannot recompute the original data. Below, we show an application of kernel PCA to the helix and demonstrate its ability to identify the one-dimensional structure of the helix and its sensitivity to kernel selection and tuning.

Example 5.1 (Kernel PCA applied to the Helix)**.** First, we show the kPCA variances for three different kernels and tuning parameters. The data are regularly spaced points along the helix.



From these graphs, one would infer very different lower dimensional choices depending on the kernel and parameters. The polynomial kernel provides the most robust estimate of the one-dimensional nature of the data.

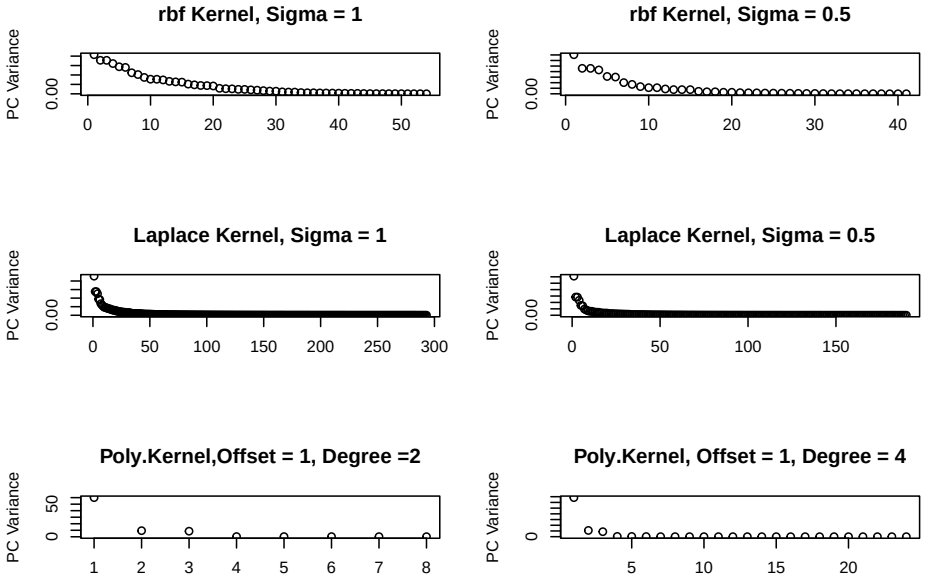
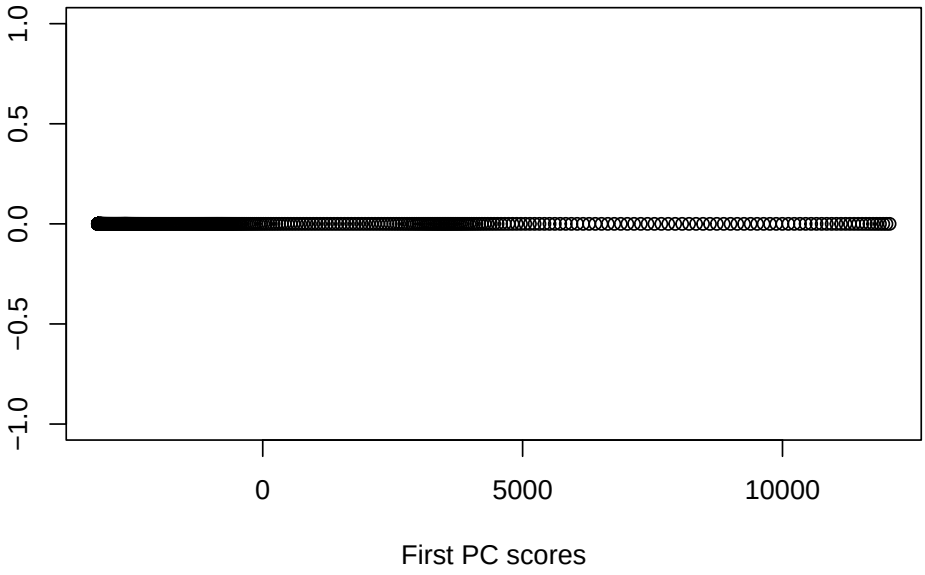


Figure 5.1: kPCA Variances for different Kernels

Below, we show the recovered one-dimensional coordinates for the polynomial kernel with offset 1 and degree 4 shown below, which is good, but do not quite reflect the equal spaced nature of the points.



As the preceding example demonstrates, kernel PCA can identify nonlinear structure, but is quite sensitive to kernel selection and tuning. More advanced implementations make use of cross-validation to aid in the selection and tuning of the kernel [**?**, **?**].

5.2 Exercises

Chapter 6

Manifold Learning

6.1 The Manifold Hypothesis

The manifold hypothesis is the central assumptions of modern nonlinear dimension reduction methods and can be stated as follows.

Definition 6.1 (Manifold Hypothesis)**.** The observed data $\vec{x}_1, \ldots, \vec{x}_N \in \mathbb{R}$ are concentrated on or near a manifold of intrinsic dimension $t \ll d$.

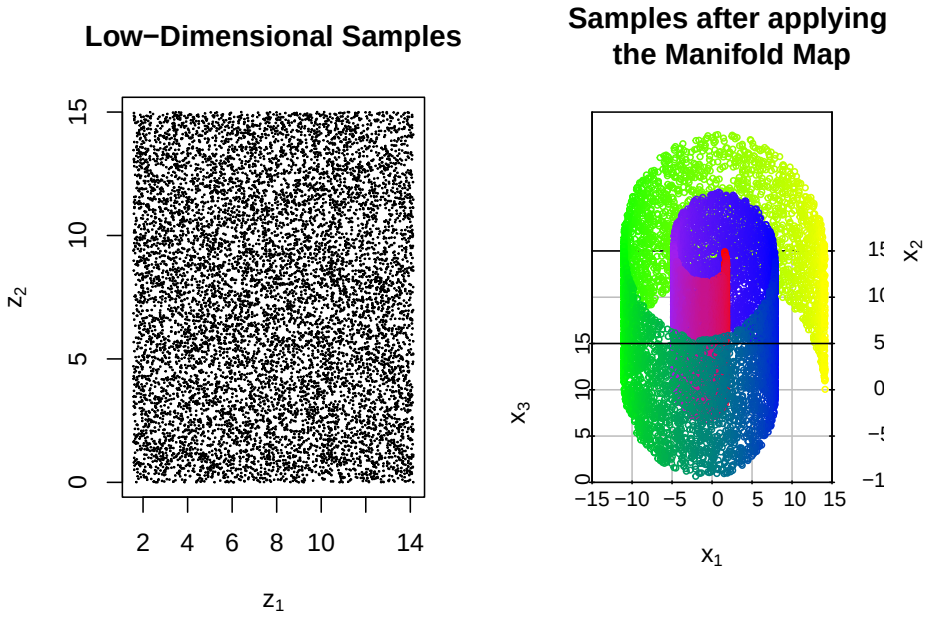
Manifolds are subspaces which could be linear but typically are not. In the following sections, we will provide more background on the mathematical foundation of manifold including important properties and assumptions used by various methods of nonlinear dimension reduction. For now, let's discuss a simple mechanism for generating data on a manifold.

Assume there are points $\vec{z}_1, \ldots, \vec{z}_N \in A \subset \mathbb{R}^k$ which are *iid* random samples from some probability distribution. These points are (nonlinearly) mapped into a higher dimensional space $\mathbb{R}^d$ by a smooth map Ψ giving data $\vec{x}_i = \Psi(\vec{z}_i)$ for $i = 1, \ldots, N$. Hereafter, we refer to Ψ as the manifold map. In this setting, we are only given $\vec{x}_1, \ldots, \vec{x}_N$, and we want to recover the lower-dimensional $\vec{z}_1, \ldots, \vec{z}_N$. If possible, we would also like recover Ψ and Ψ^{-1} and in the most ideal case, the sampling distribution that generated the lower-dimensional coordinates $\vec{z}_1, \ldots, \vec{z}_N$.

Example 6.1 (Mapping to the Swiss Roll)**.** Let $A = (\pi/2, 9\pi/2) \times (0, 15)$. We define the map $\Psi : A \to \mathbb{R}^3$ as follows

$$\Psi(\vec{z}) = \Psi(z_1, z_2) = \begin{bmatrix} z_1 \sin(z_1) \\ z_1 \cos(z_1) \\ z_2 \end{bmatrix}$$

Below we show $N = 10^4$ *iid* samples which are drawn uniformly from A. We then show the resulting observations after applying map Ψ to each sample.

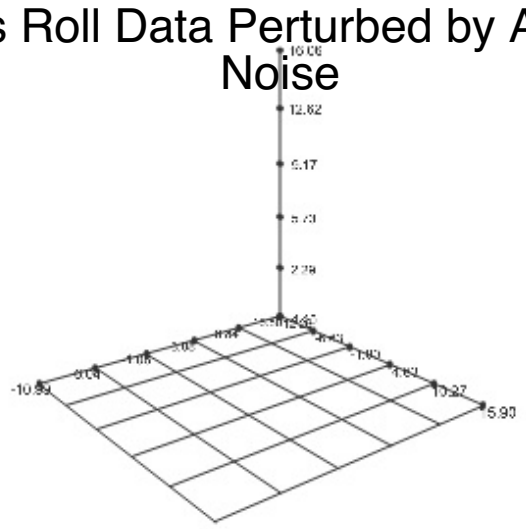


We may also consider the more complicated case where the observations are corrupted by additive noise. In this setting, the typical assumption is that the noise follows after the manifold map so that our data are

$$\vec{x}_i = \Psi(\vec{z}_i) + \vec{\epsilon}_i, \qquad i = 1, \ldots, N$$

for some *iid* noise vectors $\{\vec{\epsilon}_i\}_{i=1,\ldots,N}$.

Example 6.2 (Swiss Roll with Additive Gaussian Noise)**.** Here, we perturb the observations in the preceding example with additive $\mathcal{N}(\vec{0}, 0.1\mathbf{I})$ noise.



In addition to the goals in the noiseless case, we may also add the goal of learning the noiseless version of the data which reside on a manifold.

However, there are a number of practical issues to this setup. First, the dimension, t, of the original lower-dimensional points is typically unknown. Similar to previous methods, we could pick a value of t with the goal of visualization, base our choice off of prior knowledge, or run our algorithms different choices of t and compare the results. More advanced methods for estimating the true value of k are an open area of research [**?**].

There is also a issue with the uniqueness problem statement. Given only the high dimensional observations, there is no way we could identify the original lower-dimensional points without more information. In fact, one could find an unlimited sources of equally suitable results. Here is the issue.

Let $\Phi : \mathbb{R}^k \to \mathbb{R}^k$ be some invertible function. As an example, you could think of Φ as defining a translation, reflection, rotation, or some composition of these operations. If our original observed data are $\vec{x}_i = \Psi(\vec{z}_i)$, our manifold learning algorithm could instead infer that the manifold map is $\Psi \circ \Phi^{-1}$ and the lower-dimensional points are $\Phi(\vec{z}_i)$. This is a perfectly reasonable result since $(\Psi \circ \Phi^{-1} \circ)\Phi(\vec{z}_i) = \Psi(\vec{z}_i) = \vec{x}_i$ for $i = 1, \ldots, N$, which is the only result we require. Without additional information, there is little we could do to address this issue. For the purposes of visualization, however, we will typically be most interested in the relationship between the lower-dimensional points rather than their specific location or orientation. As such, we need not be concerned about a manifold learning algorithm that provides a translated or

rotated representation of $\vec{z}_1, \ldots, \vec{z}_N$. More complicated transformations of the lower-dimensional coordinates are of greater concern and may be addressed through additional assumptions about the manifold map Ψ.

In the following sections, we will review a small collection of different methods which address the manifold learning problem. Many of the methods are motivated based on important concepts from differential geometry, the branch of mathematics focused on manifolds. Many of the details of differential geometry are beyond the scope of this book, so we will focus on a few key ideas here. For the more mathematically-minded reader, the following texts (REFERENCES) make great references.

6.2 A brief primer on manifolds and differential geometry

In the previous sections, we have focused on methods which seek to approximate our data through a linear combination of feature vectors. As we have seen, the resulting approximations live on linear (or affine) subspaces in the case of PCA and SVD and positive spans or convex combinations in the case of NMF. While our data may exhibit some low-dimensional structure, there is no practical reason to expect such behavior to be inherently linear. In the resulting sections, we will explore methods which consider **nonlinear** structure and assume the data reside on or near a manifold. Such methods are referred to as nonlinear dimension reduction or manifold learning. Critical to this discussion is the notion of a manifold.

Definition 6.2 (Informal Definition of a Manifold)**.** A manifold is a (topological) space which locally resembles Euclidean space. Each point on a k-dimensional manifold has a neighborhood that can be mapped continuously to $\mathbb{R}^k$.

To guide your intuition, think of a manifold as a smooth, possibly curved surface. Here are a few examples.

Example 6.3 (Examples of Manifolds)**.** Add line, sphere, plane, and S

And here is an example of something which isn't a manifold.

Example 6.4 (Non-manifold)**.** figure 8

6.3 Isometric Feature Map (ISOMAP)

6.3.1 Introduction

The first manifold learning method we are going to cover is the Isometric Feature Map (ISOMAP), originally published by Tenenbaum, de Silva, and Langford in 2000 [**?**]. As suggested by the name, we will see that the assumption of isometry is central to this method. ISOMAP combines the major algorithmic features of

PCA and MDS — computational efficiency, global optimality, and asymptotic convergence guarantees. Thanks to these extraordinary features, ISOMAP is capable of learning a broad class of nonlinear manifolds.

6.3.2 Key Definitions

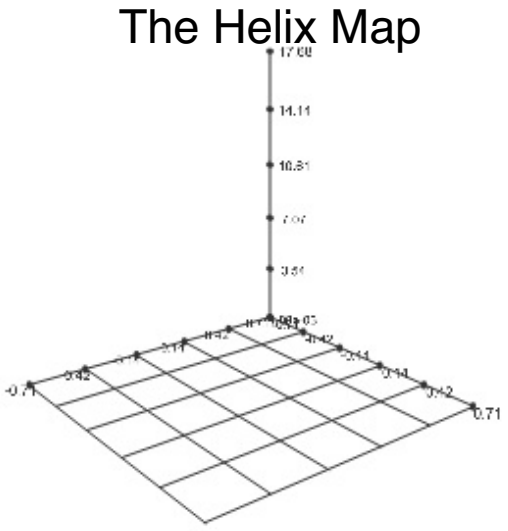
Different notions of pointwise distance

Prior to discussing the ISOMAP algorithm, let's briefly discuss the notion of isometry through an example which motivates different notions of distance between two points.

Example 6.5 (Distance between points on a Helix)**.** Consider the helix map $\Psi : \mathbb{R} \to \mathbb{R}^3$ given by the formula

$$\Psi(t) = \begin{bmatrix} \frac{1}{\sqrt{2}}\cos(t) \\ \frac{1}{\sqrt{2}}\sin(t) \\ \frac{1}{\sqrt{2}}t \end{bmatrix} \tag{6.1}$$

Below, we show the result of applying the Helix map to each point in the interval $(0, 25)$. Let's focus on two points $\vec{x}_1 = \Psi(2\pi) = (1/\sqrt{2}, 0, \sqrt{2}\pi)^T$ and $\vec{x}_2 = \Psi(4\pi) = (1/\sqrt{2}, 0, 2\sqrt{2}\pi)^T$ in particular which are shown as large black dots in the figure below.



There are a few different ways we could measure the distance between the two black points. The first approach would be to ignore the helix (manifold) structure

viewing them as vectors in $\mathbb{R}^3$ and directly measure their **Euclidean distance** which gives

$$\|\vec{x}_1 - \vec{x}_2\| = \sqrt{2}\pi.$$

However, we also know that these points are images of the one-dimensional coordinate $z_1 = 2\pi$ and $z_2 = 4\pi$ respectively. Thus, we could also consider the Euclidean distance of the lower-dimemsional coordinates which is $|2\pi - 4\pi| = 2\pi$, which notably differs from the Euclidean distance.

A third option is to return to the three-dimensional representation but to also account for the manifold structure when considering distance. Recall Euclidean distance gives the length of the shortest, **straightline** path connecting the two points. Instead, let's restrict ourselves to only those paths which stay on the helix (manifold). You may correctly conclude that the curve starting at $\Psi(2\pi)$, rotating up the helix one rotation, and ending at $\Psi(4\pi)$ is the shortest such path. Fortunately, computing arc-length is relatively friendly in this example since Ψ already parameterizes the path connecting these two points. The arc-length is then

$$\int_{2\pi}^{4\pi} \left\| \frac{d\Psi}{dt} \right\| dt = \int_{2\pi}^{4\pi} dt = 2\pi.$$

Jumping slightly ahead, we then say the manifold distance between $\Psi(2\pi)$ and $\Psi(4\pi)$ is 2π. Importantly, the manifold distance coincides exactly with the Euclidean distance between the **lower-dimensional** coordinates. In fact, for any two points, s and t, on the real line their Euclidean distance, $|s - t|$ will be the same as the manifold distance between $\Psi(s)$ and $\Psi(t)$. Thus, the helix map Ψ above serves as our first example of an isometric (distance preserving) map.

We may generalize this idea to any smooth manifold to define a new notion of distance. Given a manifold $\mathcal{M}$, we define the manifold distance function $d_{\mathcal{M}} : \mathcal{M} \times \mathcal{M} \to [0, \infty)$ as follows

Definition of Manifold Distance ::: {.definition #def-manifold-dist name="Manifold Distance Function"} Given two points $\vec{x}$ and $\vec{y}$ on a smooth manifold, $\mathcal{M}$, let $\Gamma(\vec{x}, \vec{y})$ be the set of all piecewise smooth curves connecting $\vec{x}$ and $\vec{y}$ constrained to stay on $\mathcal{M}$. Then, we define the manifold distance to be

$$d_{\mathcal{M}}(\vec{x}, \vec{y}) = \inf_{\gamma \in \Gamma(\vec{x}, \vec{y})} L(\gamma) \tag{6.2}$$

where $L(\gamma)$ is the arclength of γ.
:::

As we reviewed above, the helix example with the arclength formula is one example of a manifold and distance function. Additional examples of a manifold and manifold distance include,

- Euclidean space $\mathbb{R}^d$ where standard Euclidean distance gives the manifold distance.
- The sphere in $\mathbb{R}^3$ which is a two-dimensional manifold. Its manifold distance is also called the Great Circle Distance.

We may now define the notion of isometry which is a central assumption of ISOMAP.

Definition of Isometry ::: {.definition #def-isometry name="Isometry"} Let $\mathcal{M}_1$ be a manifold with distance function $d_{\mathcal{M}_1}$ and let $\mathcal{M}_2$ be a second manifold with distance function $d_{\mathcal{M}_2}$. The mapping $\Psi : \mathcal{M}_1 \mapsto \mathcal{M}_2$ is an isometry if

$$d_{\mathcal{M}_1}(x, y) = d_{\mathcal{M}_2}\left(\Psi(\vec{x}), \Psi(\vec{y})\right) \qquad \text{for all } \vec{x}, \vec{y} \in \mathcal{M}_1.$$

:::

For the purposes of ISOMAP, we will think of $\mathcal{M}_1$ as some subset of a $\mathbb{R}^k$ for k small where we measure distances using the Euclidean norm. Then $\mathcal{M}_2$ will be a k-dimensional manifold in $\mathbb{R}^d$ containing our data. Our first assumption is that the manifold mapping Ψ is an isometry. Unfortunately, in practice we do not know the manifold nor will we have a method for parameterizing curves on the manifold to compute distances.

6.3.3 Algorithm

Instead, ISOMAP makes use of a data-driven approach to estimate the manifold distance between points following a three-step procedure.

1) Construct Weighted Neighborhood Graph:

MDS uses Euclidean distance to measure pairwise distance between points $\vec{x}_i$ and $\vec{x}_j$ (data points in space $\mathcal{M}_2$), while ISOMAP uses the geodesic distance in order to reveal the underlying manifold structure. However, when the data points in the high dimensional space $\mathcal{M}_2$ have a manifold structure, usually the Euclidean pairwise distance is quite different from their pairwise geodesic distance. Fortunately, for small distances on a smoothly embedded manifold, the geodesic path between two close-by points lies nearly flat in the ambient space. So, the length of this path will be very close to the straight line (Euclidean) distance between those points in the ambient space.

The key intuition is that as the density of data points on the manifold increases (i.e., points get closer and closer), the straight line segment in the ambient space connecting two neighboring points becomes a better and better approximation of the shortest path between those points on the manifold. In the limit of the density going to infinity, these distances converge.

Let's elucidate this concept with two illustrative examples. Firstly, imagine a two-dimensional surface, like a Swiss Roll, situated within a three-dimensional space. For an ant journeying across the Swiss Roll, the vast size difference means its immediate surroundings appear flat. From its perspective, the distance between its consecutive steps closely mirrors the distance a human might measure (Euclidean distance) – both virtually equating to the roll's geodesic distance. For a larger-scale analogy, think of Earth. Suppose extraterrestrial beings possessed technology allowing them to traverse straight through Earth's crust and mantle,

thus following the shortest Euclidean path. Their journey from Los Angeles to New York might save them hundreds of miles compared to humans. However, when moving between closer landmarks, such as the Science Center to the Smith Center, their advantage diminishes.

As a result, when it comes to the measurement of geodesic distance, it is reasonable to only look at those data points that are close to each other. First, calculate all the pairwise Euclidean distance $d_{ij} = ||\vec{x}_i - \vec{x}_j||_2$, then determine which points are neighbors on the manifold by connecting each point to Either (i) All points that lie within a ball of radius ϵ of that point; OR (ii) all points which are K-nearest neighbors with it. (Two different criteria, K and ϵ are tuning parameters)

According to this rule, a weighted neighborhood graph $G = G(V, E)$ can be built. The set of vertices (data points in space $\mathcal{M}_2$): $V = \{\vec{x}_1, \ldots, \vec{x}_N\}$ are the input data points, and the set of edges $E = \{e_{ij}\}$ indicate neighborhood relationships between the points. $e_{ij} = d_{ij}$ if (i) $||\vec{x}_i - \vec{x}_j||_2 \leq \epsilon$; OR (ii) $\vec{x}_j$ is one of the K-nearest neighbors of $\vec{x}_i$, otherwise $e_{ij} = \infty$. Sometimes, the tuning of ϵ (or K) is quite decisive in the output of ISOMAP, we will explain this later with a simulation example.

2) Compute graph distances

In this step, we want to estimate the unknown true geodesic distances $\{d_{ij}^{\mathcal{M}}\}$ between all pairs of points with the help of the neighborhood graph G we have just built. We use the graph distances $\{d_{ij}^{\mathcal{G}}\}$— the shortest distances between all pairs of points in the graph G to estimate $\{d_{ij}^{\mathcal{M}}\}$. For $\vec{x}_i$ and $\vec{x}_j$ that are not connected to each other, we try to find the shortest path that goes along the connected points on the graph. Following this particular sequence of neighbor-to-neighbor links, the sum of all the link weights along the path is defined as $\{d_{ij}^{\mathcal{G}}\}$. In other words, we use a number of short Euclidean distances (representing the local structure of the manifold) to approximate the geodesic distance $\{d_{ij}^{\mathcal{M}}\}$.

This path finding step is usually done by Floyd-Warshall algorithm, which iteratively tries all transit points k and find those that $\tilde{d}_{ik} + \tilde{d}_{kj} < \tilde{d}_{ij}$, and updates $\tilde{d}_{ij} = \tilde{d}_{ik} + \tilde{d}_{kj}$ for all possible combination of i, j. The algorithm works best in dense neighboring graph scenario, with a computational complexity of $O(n^3)$.

The theoretical guarantee of this graph distance computation method is given by Bernstein et, al.[**?**] one year after they first proposed ISOMAP in their previous paper. They show that asymptotically (as $n \to \infty$), the estimate $d^{\mathcal{G}}$ converges to $d^{\mathcal{M}}$ as long as the data points are sampled from a probability distribution that is supported by the entire manifold, and the manifold itself is flat.

The distance matrix Δ can be expressed as:

$$\Delta_{ij} = d_{ij}^{\mathcal{G}}$$

Simulation Example

Here we provide a randomly generated Neighborhood Graph for six data points, it uses the K-nearest neighbor criteria (can easily tell this since the matrix is not symmetric, $K = 2$)

```
# Define the matrix
matrix <- matrix(c(
  0,   3,  4,   Inf, Inf, Inf,
  7,   0,  Inf, 2,   Inf, Inf,
  6,   Inf,0,   Inf, 7,   Inf,
  Inf, 5,  Inf, 0,   Inf, 10,
  Inf, Inf,8,   Inf, 0,   13,
  Inf, Inf,Inf, 9,   14,  0
), byrow = TRUE, nrow = 6)
print(matrix)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    3    4  Inf  Inf  Inf
## [2,]    7    0  Inf    2  Inf  Inf
## [3,]    6  Inf    0  Inf    7  Inf
## [4,]  Inf    5  Inf    0  Inf   10
## [5,]  Inf  Inf    8  Inf    0   13
## [6,]  Inf  Inf  Inf    9   14    0
```

Shown below is the implementation of Floyd-Warshall algorithm in R. As you can see from the three for loops, its computation complexity is $O(n^3)$.

```
# Adjusting the matrix to set d_ij and d_ji to the smaller value
n <- dim(matrix)[1]

for (i in 1:n) {
  for (j in 1:n) {
    if (i != j && is.finite(matrix[i, j]) && is.finite(matrix[j, i])) {
      min_val <- min(matrix[i, j], matrix[j, i])
      matrix[i, j] <- min_val
      matrix[j, i] <- min_val
    }
  }
}

# Floyd-Warshall Algorithm
floyd_warshall <- function(mat) {
  n <- dim(mat)[1]
  dist <- mat

  for (k in 1:n) {
    for (i in 1:n) {
      for (j in 1:n) {
```

```
        dist[i, j] <- min(dist[i, j], dist[i, k] + dist[k, j])
      }
    }
  }

  return(dist)
}

# Get the result
result <- floyd_warshall(matrix)

# Print the result
print(result)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    3    4    5   11   14
## [2,]    3    0    7    2   14   11
## [3,]    4    7    0    9    7   18
## [4,]    5    2    9    0   16    9
## [5,]   11   14    7   16    0   13
## [6,]   14   11   18    9   13    0
```

3) Applying MDS to Δ

As mentioned before, ISOMAP can be viewed as the application of classical MDS in non-linear case. As a result, the reconstruction of $\{\vec{z}_i\}$ in the k dimensional $\mathcal{M}_1$ follows similar steps as that of classical MDS. The main goal is to preserve the geodesic distance of the manifold in $\mathcal{M}_2$ as much as possible.

Without any additional information, there are infinite $\{\vec{z}_i\}$ that can be viewed as the optimal solution. For some invertible function $\Phi : \mathbb{R}^k \to \mathbb{R}^k$, a new manifold mapping $\Psi \circ \Phi^{-1}$ can be constructed. $\vec{x}_i = \Psi \circ \Phi^{-1}(\Phi(\vec{z}_i))$, which proofs that $\{\Phi(\vec{z}_i)\}$ is equivalent to $\{\vec{z}_i\}$ when it comes to the reconstruction of the lower dimensional configuration.

Without loss of generality, we assume that $\{\vec{z}_i\}$ are actually centered. So the distance matrix of $\{\vec{z}_i\}$ can be expressed as $B = Z^T Z$, so that $B_{ii} = ||z_i||_2^2$ and $B_{ij} = {z_i}^T z_j$.

The embedding vectors $\{\hat{z}_i\}$ (estimate of points in lower dimensional feature space $\mathcal{M}_1$) are chosen in order to minimize the objective function:

$$(\sum ||\vec{z}_i - \vec{z}_j||_2 - \Delta_{ij})^2$$

Following the same procedure explained in classical MDS chapter, we can compute

each entry of B:

$$B_{ij} = -\frac{1}{2}\Delta_{ij}^2 + \frac{1}{d}\sum_{i=1}^{d}\Delta_{ij}^2 + \frac{1}{d}\sum_{j=1}^{d}\Delta_{ij}^2 - \frac{1}{2d^2}\sum_{i=1}^{d}\sum_{j=1}^{d}\Delta_{ij}^2$$

To express it in matrix form, it is actually, $B = -\frac{1}{2}H\Delta H$, where $H = I_n - \frac{1}{n}\not{1}\not{1}^T$.

The next step is just a PCA problem. Implement eigen decomposition on matrix B, $B = U\Lambda U^T = (\Lambda^{1/2}U)^T(\Lambda^{1/2}U)$, then arrange the singular value in descending order, find the first k' ones. We acquire $\Lambda_{k'}$ and $U_{k'}$.

$$(\hat{z}_1|\hat{z}_2|\ldots|\hat{z}_N) = \Lambda_{k'}U_{k'}$$

Since we don't know the dimension of the underlying feature space, here k' is a tuning parameter. Usually, we use a scree plot (k' against the sum of the omitted eigenvalues) and find the elbow point.

6.3.4 Limitations of ISOMAP

Though ISOMAP is a powerful manifold learning method that works well under most circumstances. It still has some limitations in certain scenarios.

1) If the noises $\{\epsilon_i\}$ is not negligible, then ISOMAP may fail to identify the manifold. Also, ISOMAP is quite sensitive to the tuning parameters. To alleviate the negative impact, it's highly suggested to start with a relatively small ϵ or K, and increase them gradually.

2) When data points are sparse in certain areas or directions of the manifold, the integrity of the learned manifold structure can be compromised. The following example will clarify this notion:

3) One of the two major assumptions of ISOMAP is the convexity of the manifold, that is to say, if the manifold contains many holes and concave margins, then the result of ISOMAP will probably be not ideal.

6.4 Locally Linear Embeddings (LLEs)

6.4.1 Introduction

Locally linear embedding (LLE) is an unsupervised learning algorithm first introduced in 2000 by Sam T. Roweis and Lawrence K. Saul [**?**]. In the original four-page paper, the two authors introduced the LLE algorithm and demonstrated its effectiveness in dimensional reduction, manifold learning, and in handling real-world high-dimensional data. Unlike clustering methods for local dimensional reduction, LLE maps its inputs into a single global coordinate system of lower dimensionality, and its optimizations do not involve local minima. By exploiting the local symmetries of linear reconstructions, LLE is able to learn the global

structure of nonlinear manifolds, such as those generated by images of faces or documents of text. Thanks to its great mathematical properties and relatively low computing cost (compared to other manifold learning methods, like ISOMAP), LLE quickly became attractive to researchers after its emergence due to its ability to deal with large amounts of high dimensional data and its non-iterative way of finding the embeddings [**?**]. Compared to ISOMAP and some other previous manifold learning methods, LLE is computationally simpler and can give useful results on a broader range of manifolds [**?**].

For dimensional reduction, most methods introduced before LLE need to estimate pairwise distances between even two remote data points, no matter it is the simple Euclidean distance (classical MDS) or more sophisticated manifold distance (ISOMAP). The underlying main idea of these methods is actually finding a configuration that recovers all pairwise distances of original data points as much as possible. LLE, however, is quite different from these previous methods as it focuses on preserving **locally linear relationships**.

6.4.2 Algorithm

LLE algorithm is actually built on very simple geometric intuitions. As explained in MDS Part, if we consider a small enough region on a manifold in D dimensional space, in most cases, it can be regarded as a d dimensional hyperplane ($d \ll D$). LLE also makes use of this intuition and assumes that the whole manifold consist of numerous d-dimensional patches that have been stitched together. Assuming that there exists sufficient data (data points are compact), it is reasonable to expect that each data point and its neighbors lie on or close to a locally linear patch of the manifold.

Following this idea, LLE approximates each data point by a weighted linear combination of its neighbors and proceeds to find a lower-dimensional configuration of data points so that the linear approximations of all data points are best preserved.

Specifically speaking, LLE algorithm consists of three steps. The initial step involves selecting a certain number of each data point's nearest neighbors based on Euclidean distance. Following this, the second step calculates the optimal reconstruction weights for each point using its nearest neighbors. The final step carries out the embedding while maintaining the local geometry depicted by the reconstruction weights.

6.4.2.1 Construct Neighborhood Graph

This step is actually very similar to that of ISOMAP. The process of finding neighbors in LLE is typically conducted using grouping methods like k-nearest neighbors (KNN) or selecting neighbors within a fixed radius ball (ϵ-neighborhoods), based on the Euclidean distance for each data point, in the provided data set. The KNN method is predominantly utilized for its straightforwardness and ease of implementation. The following explanations are based on KNN method.

Denote N data points in original D dimensional space as $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N \in \mathbb{R}^D$. For a point $\vec{x}_i, \quad 1 \leq i \leq N$, its neighbor set is defined as $N_i^k \subseteq \{1, 2, 3, \ldots, i-1, i+1, \ldots, N\}$, where N_i^k can also be called as the indices of k nearest neighbors of $\vec{x}_i$. The tuning parameter k is chosen small enough so that the patch around $\vec{x}_i$ is flat. However, k should also be strictly larger than d so as to let the algorithm work.

As we can tell from these, LLE works well only if data points are dense and hopefully evenly distributed, which will be explained in detail in later examples. The parameter tuning of the appropriate number of neighbors, k, faces challenges of complexity, non-linearity, and diversity of high-dimensional input samples. A larger k value might cause the algorithm to overlook or even lose the local nonlinear features on the manifold. This issue is exacerbated as neighbor selection, typically based on Euclidean distance, can result in distant neighbors when considering the intrinsic geometry of the data, akin to a short circuit. Conversely, an overly small k value may lead the LLE algorithm to fragment the continuous manifold into isolated local pieces, losing global characteristics.

6.4.2.2 Reconsruct with Linear Weights

As put before, we try to reconstruct each $\vec{x}_i$ using an almost convex weighted combination of its neighbors. The respective weights of all its neighbors $\vec{x}_j, \ j \neq i$ for each $\vec{x}_i$ is quite essential in the later reconstruction of the underlying intrinsic configuration, as we consider these weights to remain invariant before and after mapping.

To explain it in mathematical formulas, the approximate of $\vec{x}_i$: $\tilde{x}_i$ is defined as $\tilde{x}_i = \sum_{j=1}^{N} w_{ij}\vec{x}_j$. There are two constraints for this formula: First, $w_{ij} \equiv 0$, if $j \notin N_i^k$ (consistent with the assumption of k nearest neighbors); Second, the sum of weights for each $\vec{x}_i$ is always zero, i.e., $\sum_{j=1}^{N} w_{ij} = 1$.

Then, the problem of finding the optimal $w_{ij}, \ 1 \leq i, j \leq N$ is equivalent to solving the following constrained Least Squares problem for $\forall 1 \leq i \leq N$:

(1)

$$\begin{aligned} \min \quad & \left\| \vec{x}_i - \sum_{j \in N_i^k} w_{ij}\vec{x}_j \right\|^2 \\ \text{s.t.} \quad & \sum_{j \in N_i^k} w_{ij} = 1. \end{aligned}$$

It is worth noting that the weights can be negative theoretically, though in practice, we don't expect that to happen.

Invariance to Rotation, Rescaling and Transaction

Define $\epsilon(w) = \sum_{i=1}^{N} \left\| \vec{x}_i - \sum_{j \in N_i^k} w_{ij}\vec{x}_j \right\|^2$, which is the cost function.

1) $\epsilon(w)$ is unchanged by rotation or rescaling by common factor

Actually $\sum_{i=1}^{N} \left\| a\mathrm{U}\vec{x}_i - \sum_{j\in N_i^k} w_{ij} a\mathrm{U}\vec{x}_j \right\|^2 = a^2\epsilon(w)$, where a is a non-zero scaler and U is an orthonormal matrix.

2) $\epsilon(w)$ is unchanged by transactions

Thanks to the constraint that $\sum_{j=1}^{N} w_{ij} = 1$, for any transaction $\vec{x}_i \to \vec{x}_i + \vec{y}$, the cost function does not change.

$$\sum_{i=1}^{N} \left\| (\vec{x}_i + \vec{y}) - \sum_{j\in N_i^k} w_{ij}(\vec{x}_j + \vec{y}) \right\|^2 = \sum_{i=1}^{N} \left\| \vec{x}_i - \sum_{j\in N_i^k} w_{ij}\vec{x}_j \right\|^2 = \epsilon(w)$$

From the expressions, we develop a strategy that optimizes one row of matrix w at a time. Now let's try to rewrite $\epsilon(\vec{w}_i) = \left\| \vec{x}_i - \sum_{j\in N_i^k} w_{ij}\vec{x}_j \right\|^2$.

$$\begin{aligned}
\epsilon(\vec{w}_i) &= \left\| \vec{x}_i - \sum_{j\in N_i^k} w_{ij}\vec{x}_j \right\|^2 && (6.3)\\
&= \left[\sum_{j=1}^{N} w_{ij}(\vec{x}_i - \vec{w}_j) \right]^T \left[\sum_{l=1}^{N} w_{il}(\vec{x}_i - \vec{w}_l) \right]^T && (6.4)\\
&= \sum_{j=1}^{N}\sum_{l=1}^{N} w_{ij}w_{il}(\vec{x}_i - \vec{x}_j)^T(\vec{x}_i - \vec{x}_l) && (6.5)\\
&= \vec{w}_i^T G_i \vec{w}_i && (6.6)
\end{aligned}$$

$\vec{w}_i^T = (w_{i1}, w_{i2}, \ldots w_{iN})$ is the i^{th} row of W. Here $G_i \in \mathbb{R}^{N\times N}$, where entry $G_i(j.l)$, $1 \le j, l \le N$ can be represented as:

$$G_i(j,l) = \begin{cases} (\vec{x}_i - \vec{x}_j)^T(\vec{x}_i - \vec{x}_l) & j, l \in N_i^k \\ 0 & j \text{ or } l \notin N_i^k \end{cases}$$

The (j, l) entry of G_i is actually the inner product of $\vec{x}_j$ and $\vec{x}_l$ when centered around $\vec{x}_i$. From this expression, we know that actually G_i is a sparse matrix and can be reduced to a compact matrix $\tilde{G}_i \in \mathbb{R}^{k\times k}$ that eliminates those empty columns and rows.

$$\begin{aligned}
\tilde{G}_i &= (\vec{x}_{i[1]} - \vec{x}_i, \ldots, \vec{x}_{i[k]} - \vec{x}_i)^T(\vec{x}_{i[1]} - \vec{x}_i, \ldots, \vec{x}_{i[k]} - \vec{x}_i) && (6.7)\\
&= Q_i^T Q_i && (6.8)
\end{aligned}$$

where [1] denotes the first entry in N_i^k. So $\tilde{G}_i$ is actually a real symmetric and positive semi-definite matrix.

Now let's go back to deal with the optimization function — Equation 1 can be solved with Lagrange multiplier given that it has only equality constraints. (More details about the use of Lagrange multiplier can be found in [Lagrange multiplier]https://en.wikipedia.org/wiki/Lagrange_multiplier)

Optimizing Equation 1 is equivalent to minimizing (for $\forall 1 \leq i \leq N$)

$$f(\vec{w}_i, \lambda) = \vec{w}_i^T G_i \vec{w}_i - \lambda(\vec{w}_i^T \mathbf{1}_k - 1)$$

which has the result:

$$\vec{w}_i^\star = \frac{\tilde{G}_i^- \mathbf{1}_k}{\mathbf{1}_k^T \tilde{G}_i^- \mathbf{1}_k}$$

Complement:

As discussed before, we can only proof that $\tilde{G}_i$ is positive semi-definite, however, we cannot ensure that it is positive definite, which means $\tilde{G}_i$ is not necessarily invertible. That is why we use the generalize inverse sign here. In practice, it can be done through performing SVD on $\tilde{G}_i$ and select the first few large singular values and eliminate the rest. Then when computing $\tilde{G}_i^-$, just do the reciprocal of these retained singular values.

Actually $\tilde{G}_i$ only has d (the intrinsic original dimension if you forget) relatively large eigenvalues. The rest are either very small or zeros. So it is very likely that $\tilde{G}_i$ is singular, making the computation result highly unstable. In {**?**}, the authors proposed to address this issue through regularizing $\tilde{G}_i$.

$$\tilde{G}_i \leftarrow \tilde{G}_i + \left(\frac{\Delta^2}{k}\right) \mathrm{Tr}\left(\tilde{G}_i\right) \mathbf{I}$$

Here $Tr(\tilde{G}_i)$ denotes the trace of $\tilde{G}_i$ and $\Delta \ll 1$.

6.4.2.3 Embedding

In the previous step, we have recovered the optimal weight matrix

$$\mathbf{W} = \begin{pmatrix} \vec{w}_1^T \\ \vdots \\ \vec{w}_N^T \end{pmatrix}$$

The optimal weights $\mathbb{W}$ reflects local, linear geometry around each $\vec{x}_i$, thus if the configuration $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N \in \mathbb{R}^d$ are the lower dimensional representation, they should also "match" the local geometry.

Following this idea, the objective function is:

(2)

$$\begin{aligned}
&\underset{\mathbf{Y}}{\operatorname{argmin}} \sum_{i=1}^{N} \left\| \vec{y}_i - \sum_{j=1}^{N} w_{ij} \vec{y}_j^T \right\|^2 \\
=&\underset{\mathbf{Y}}{\operatorname{argmin}} \sum_{i=1}^{N} \left\| \sum_{j=1}^{N} w_{ij} (\vec{y}_i - \vec{y}_j)^T \right\|^2 \\
=&\underset{\mathbf{Y}}{\operatorname{argmin}} \|\mathbf{Y} - \mathbf{W}\mathbf{Y}\|_F^2 \\
=&\underset{\mathbf{Y}}{\operatorname{argmin}} \|(\mathbf{I}_N - \mathbf{W})\mathbf{Y}\|_F^2 \\
=&\underset{\mathbf{Y}}{\operatorname{argmin}} \operatorname{Tr}\left[\mathbf{Y}^T (\mathbf{I}_N - \mathbf{W})^T (\mathbf{I}_N - \mathbf{W})\mathbf{Y}\right]
\end{aligned}$$

where $\mathbf{Y} = (\vec{y}_1 | \vec{y}_2 | \dots | \vec{y}_N)^T$

There are two constraints:

a) $\mathbf{1}_N^T \mathbf{Y} = \vec{0}$. This forces $\vec{y}$ to be centered

b) $\frac{1}{N}\mathbf{Y}^T\mathbf{Y} = \mathbf{I}_d$. This fixes rotation and scaling.

Key Observation

Considering the final expression of Equation 2, the optimization function is now equivalent to finding $\vec{y}_i$s that minimizes $\mathbf{Y}^T(\mathbf{I}_N - \mathbf{W})^T(\mathbf{I}_N - \mathbf{W})\mathbf{Y}$.

Here we introduce $\mathbf{M} = (\mathbf{I}_N - \mathbf{W})^T(\mathbf{I}_N - \mathbf{W})$, which is a positive semi-definite matrix. Since $\mathbf{M}\mathbf{1}_N = (\mathbf{I} - \mathbf{W})^T(\mathbf{1}_N - \mathbf{W}\mathbf{1}_N) = \vec{0}$, $\mathbf{1}_N$ is an eigen-vector of $\mathbf{M}$ with eigenvalue zero.

$$\mathbf{Y}^T(\mathbf{I}_N - \mathbf{W})^T(\mathbf{I}_N - \mathbf{W})\mathbf{Y} = \mathbf{Y}^T\mathbf{M}\mathbf{Y}$$

From constraint (b), we know that columns of $\mathbf{Y}$ are orthogonal to each other. As a result, this whole problem can be simplified to finding the eigen-vectors of $\mathbf{M}$ with the smallest eigenvalues.

Compute eigen-vectors with the smallest $d+1$ eigenvalues $0 = \lambda_1 \leq \lambda_2 < \cdots < \lambda_{d+1}$, eliminate $\mathbf{1}_N$ (the first one). The remaining d vectors are respectively $\vec{v}_2, \vec{v}_3, \dots \vec{v}_{d+1} \in \mathbb{R}^N$. So $\mathbf{Y} = (\vec{v}_2 | \vec{v}_2 | \dots | \vec{v}_{d+1})$, we successfully recover the corresponding $\vec{y}_1, \vec{y}_2, \dots, \vec{y}_N \in \mathbb{R}^d$.

An illustration of the algorithm

In the original paper [**?**], the authors provide a very intuitive plot that summarizes the above three steps.

Parameter Tuning

There are two parameters to tune in LLE, i.e. (the number of neighbors: k; the dimension of the recovered configuration: d).

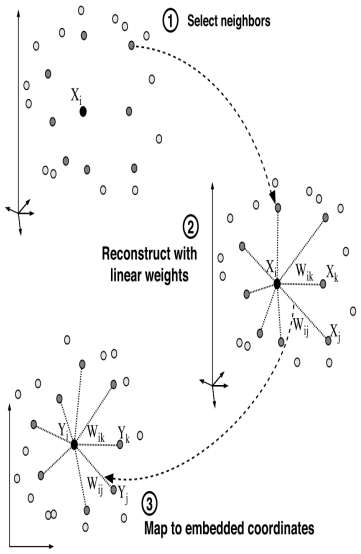


Figure 6.1: LLE_illustration

1) For selection of d, we usually use a reverse scree plot and find the elbow point. It is worth noting that we are choosing the smallest $d+1$ eigenvalues and compute their corresponding eigen-vectors here. Since the eigen-vectors and eigenvalues of a particular matrix is super sensitive to any sort of noises or perturbations, especially for those small eigenvalues, it is hard to accurately derive the corresponding eigen-vectors $\vec{v}_2, \ldots, \vec{v}_{d+1}$. This is called ill-conditioned eigen-problem.

2) Choose the optimal k

LLE seeks to preserve local structure through nearest neighbor connections. This is the key point to LLE. As a result, we may use the neighbor set of the original $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N \in \mathbb{R}^D$ and $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N \in \mathbb{R}^d$ as a criteria.

As explained before, we use N_i^k to denote the indices of k-nearest neighbors to $\vec{x}_i$. Similarly, we can also use V_i^k to denote the indices of k-nearest neighbors to $\vec{y}_i$. They should be as close as possible.

So our objective function here is:

$$Q(k) = \frac{\sum_{i=1}^{N} \left| N_i^k \cap V_i^k \right|}{Nk}$$

Plot $Q(k)$ against k, select $k^\star$ where the increase of $Q(k)$ becomes negligible.

6.4.3 Strengths and Weaknesses of LLE

6.4.3.1 Strengths

1) **High Computation Efficiency**

The low computation cost of LLE algorithm may be its most shinning advantage over other manifold learning methods, and it is actually one of its biggest selling point when it was first introduced. The LLE algorithm Involves solving a sparse eigen problem, with computational complexity of roughly $O(N^2d^2 + Nd^3)$ where N is the number of data points and d is the dimension of the recovered configuration.

In comparison, ISOMAP requires computing shortest paths between all pairs of points, which is typically done using Dijkstra's or Floyd-Warshall algorithm, leading to a complexity of $O(N^2 logN)$ or $O(N^3)$ respectively. Then, it involves eigen decomposition similar to classical MDS which is $O(N^3)$.

In practice, $d \ll N$, hence the computation cost of LLE is lower than that of ISOMAP in most cases.

2) **Few parameters to tune**

There are only two parameters to tune, respectively the number of neighbors included in the map: k, and the dimensional of the original configuration: d. In addition, there exist clear methods to find the optimal k and d, as stated in the previous part. This makes LLE algorithm easy to find the optimal parameters.

6.4.3.2 Weaknesses

1) **Sensitivity to tuning parameters**

The result of LLE is quite sensitive to its two control parameters: the number of neighbors k and the dimensional of the original configuration: d.

Here we use the Swiss Roll example to illustrate this. LLE is optimal at $k = 45$. However, when $k = 40$, the recovered lower-dimensional configuration is wrong (Green points and yellow points overlap, which is not the case in Swiss Roll); and when we slightly increase k to 50, the recovered two-dimensional expression is not necessarily a rectangle.

2) **Vulnerable to sparse or unevenly-distributed samples**

The vulnerability towards sparsity and uneven distribution exists in almost all manifold learning methods, including ISOMAP, as we have illustrated in the previous section. LLE is not immune to this either. When a data set is unevenly distributed, since LLE relies on the original Euclidean distance metric, it tends to select neighbors from a singular direction where these neighbors are densely clustered. Clearly, using these selected neighbors to reconstruct the reference point results in significant redundancy in that specific direction. Concurrently, essential information from other directions or regions is not retained for the

reconstruction of the reference point. As a result, these selected neighbors are inadequate for accurately representing and reconstructing the reference point. Consequently, much of the intrinsic structure and internal features will be lost after dimension reduction using LLE.

3) **Sensitivity to noise**

LLE is extremely sensitive to noise. Even a small noise would cause failure in deriving low dimensional configurations. Justin Wang, et.al utilize various visualization examples to illustrate this drawback in their paper [**?**], you may take a look if you are interested. Various algorithms have been developed to address this issue, i.e., Robustly Locally Linear Embedding (RLLE) [**?**], and Locally Linear Embedding with Additive Noise (LLEAN) [**?**]. The former works well when outliers exist, while the latter has a satisfactory performance when the original points are distorted with noises.

6.5 Laplacian Eigenmap

The Laplacian eigenmap [**?**] is method of manifold learning with algorithmic and geometric similarities to LLEs. Like LLEs and ISOMAP, Laplacian eigenmaps make use of k-nearest neighbor relationships and the solution of an eigenvalue problem to reconstruct the low-dimensional manifold. As suggested by the name, we will be using the graph Laplacian matrix and emphasize the preservation of nearby points on the manifold making Laplacian Eigenmaps a local method with a different emphasis than LLEs.

The graph Laplacian is an important matrix representation of our data which we will revisit later when discussing spectral clustering. In practice, Laplacian eigenmaps use sparse version of the graph Laplacian. For now, we will briefly introduce this matrix without sparsity and an important identity relating the graph Laplacian and the loss function we will minimize when constructing our low-dimensional representation.

Given data $\vec{x}_1, \ldots, \vec{x}_N$, we are going to build a weighted graph $\mathcal{G} = (V, \mathcal{E}, \mathbf{W})$, with one node per sample and weighted, undirected edges connecting the nodes. The weights, $\mathbf{W}_{ij} \geq 0$, $1 \leq i, j \leq N$, correspond to a notion of affinity, which is typically a decreasing function of the distance between our data. Two common choices are (i) binary weights:

$$\mathbf{W}_{ij} = \begin{cases} 1 & \|\vec{x}_i - \vec{x}_j\| \leq \epsilon \\ 0 & \text{else} \end{cases}$$

and (ii) weights based on the radial basis function:

$$\mathbf{W}_{ij} = \exp\left(-\frac{\|\vec{x}_i - \vec{x}_j\|^2}{2\sigma^2}\right).$$

The symmetric matrix $\mathbf{W}$ is called the (weighted) adjacency matrix of the graph, $\mathcal{G}$, encodes of all the pairwise relationships. We always set the diagonal entries of $\mathbf{W}$ to zero to preclude self-connections in the graph.

From the adjacency matrix, we can compute the total affinity of each node (sample) to all other nodes (samples) by summing along the rows. Specifically, the ith entry of the vector $\mathbf{W}\vec{1}_N$, has the total affinity of the ith node ($\vec{x}_i$) to all other nodes (data). Using these summed affinities, we then create the graph Laplacian

$$\mathbf{L} = \mathbf{D} - \mathbf{W} \tag{6.9}$$

where $\mathbf{D}$ is a diagonal matrix with entries $\mathbf{W}\vec{1}_N$ along its diagonal. The graph Laplacian is a symmetric matrix, and while not obvious at first glance, it is also positive semidefinite thanks to the following important identity. Given any vector $\vec{y} \in \mathbb{R}^N$,

$$\vec{y}^T \mathbf{L} \vec{y} = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{N} \mathbf{W}_{ij} (y_i - y_j)^2 \tag{6.10}$$

However, it is not full rank. The previous equation shows that $\vec{1}$ is an eigenvector with eigenvalue 0.

In the previous statement we can view the entries of vector $\vec{y}$ as N separate scalars. However, we can also extend the preceding expression to include Euclidean distances between vectors $\vec{y}_1, \ldots, \vec{y}_N \in \mathbb{R}^t$ to give the following important expression

$$\sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{W}_{ij} \|\vec{y}_i - \vec{y}_j\|^2 = \frac{1}{2} tr\left(\mathbf{Y}^T \mathbf{L} \mathbf{Y}\right) \tag{6.11}$$

where $\mathbf{Y} \in \mathbb{R}^{N \times t}$ has rows $\vec{y}_1^T, \ldots, \vec{y}_N^T$. We will return to this identity and its implications for Laplacian Eigenmaps after discussing the algorithmic details of the method

6.5.1 Algorithm

6.5.1.1 Compute neighbor relationships

Fix either a number of nearest neighbors $k > 0$ or maximum distance $\epsilon > 0$. If using ϵ, then $\vec{x}_i$ and $\vec{x}_j$ (correspondingly nodes i and j in the graph) are neighbors if $\|\vec{x}_i - \vec{x}_j\| \leq \epsilon$. Alternatively, if using the nearest neighbor parameter k, then we consider $\vec{x}_i, \vec{x}_j$ to be neighbors if $\vec{x}_i$ is one the k closest points to $\vec{x}_j$ **and** $\vec{x}_j$ is one of the k closest points to $\vec{x}_i$. The construction of neighbors, like the use of pairwise distance alone, results in symmetric neighbor relationship. We then connect nodes i and j with an edge if $\vec{x}_i$ and $\vec{x}_j$ are neighbors.

6.5.1.2 Compute weights and build graph Laplacian

Nodes which are not connected immediately receive an edge weight equal to zero. For all connected nodes, we compute the edge weight

$$\mathbf{W}_{ij} = \exp\left(-\frac{\|\vec{x}_i - \vec{x}_j\|^2}{2\sigma^2}\right).$$

Here we have shown weights based on the radial basis function, which is motivated by theoretical connections to the heat kernel and an approximation of the Laplacian on the manifold $\mathcal{X}$ [**?**]. As such, this method is the default in most implementation of Laplacian eigenmaps. The parameter σ^2 does require tuning which can have a large impact on the performance of the algorithm.

When k or ϵ are small, which is typically the case in practice, the (weighted) adjacency matrix $\mathbf{W}$ will be sparse (most entries equal to 0). From this adjacency matrix, we then construct the graph Laplacian as above. The graph Laplacian built from these weights will also be sparse. By preserving only those connections between nearest points, we have only maintained the pairwise, local relationships on the manifold. We will now use $\mathbf{L}$ to construct a lower-dimensional representation of the data.

6.5.1.3 Solve generalized eigenvalue problem

Consider the loss function

$$\mathcal{L}(\vec{y}_1, \ldots, \vec{y}_N) = \sum_{i=1}^{N}\sum_{j=1}^{N} \mathbf{W}_{ij}\|\vec{y}_i - \vec{y}_j\|^2. \tag{6.12}$$

This loss function is most sensitive to large pairwise distance $\|\vec{y}_i - \vec{y}_j\|$ when $\mathbf{W}_{ij}$ is also large (our original data were close). Thus, minimizing the preceding penalty prioritizes keeping $\vec{y}_i$ and $\vec{y}_j$ close when $\vec{x}_i$ and $\vec{x}_j$ have a high affinity (weight). As a result, Laplacian eigenmaps emphasize local geometry.

Vectors $\vec{y}_1, \ldots, \vec{y}_N$ which minimizes this loss function are not unique. First, there is an issue of translation. To address this issue, we will add a constraint that $\mathbf{DY}$ is a centered data matrix, i.e. $\mathbf{Y}^T\mathbf{D}\vec{1} = \vec{0}$. We can view the matrix $\mathbf{DY}$ as a reweighting of the data matrix $\mathbf{Y}$ with higher weights $\mathbf{D}_{ii}$ for data $\vec{x}_i$ which are closer to more points. Here $\mathbf{D}$ is the diagonal matrix used in the definition of the graph Laplacian. Note that

$$\mathbf{DY} = \begin{bmatrix} \mathbf{D}_{11}\vec{y}_1^T \\ \vdots \\ \mathbf{D}_{NN}\vec{y}_N \end{bmatrix}.$$

Requiring $\mathbf{DY}$ to be centered results in configurations where those points with highest affinity a constrained close to the origin in our lower dimensional representation.

However, solving the optimization problem with this modified centering constraint is still ill-posed, namely we could take $\vec{y}_1 = \cdots = \vec{y}_N = \vec{0}$ giving a configuration which is collapsed onto the origin. In fact, given any configuration $\vec{y}_1, \ldots, \vec{y}_N$ we could decrease the loss by rescaling all of our data by some constant scalar scalar $0 < c < 1$ since $\mathbf{D}(c\mathbf{Y}) = c\mathbf{D}\mathbf{Y}$ will still be centered. To address this scaling issue and give a meaningful t-dimensional configuration, we also add the constraint $\mathbf{Y}^T\mathbf{D}\mathbf{Y} = \mathbf{I}_t$. This constraint eliminates the collapse of the t-dimensional configuration onto a $t-1$ dimensional hyperplane, and in particular eliminates the cases where the $1-$dimensional configuration collapses onto a point.

Thus, we seek a data matrix $\mathbf{Y} \in \mathbb{R}^{N\times t}$ solving the following constrained optimization problem

$$\text{argmin}_{\mathbf{Y}^T\mathbf{D}\mathbf{Y}=\mathbf{I}_t, \mathbf{Y}^T\mathbf{D}\vec{1}^T=\vec{0}} = tr(\mathbf{Y}^T\mathbf{L}\mathbf{Y}). \tag{6.13}$$

To solve this problem, we first introduce the change of variable $\tilde{\mathbf{Y}} = \mathbf{D}^{1/2}\mathbf{Y}$ so that the constraints become

$$\mathbf{Y}^T\mathbf{D}\vec{1}^T = (\mathbf{D}^{1/2}\mathbf{Y})^T\mathbf{D}^{1/2}\vec{1}^T = \tilde{\mathbf{Y}}^T\mathbf{D}^{1/2}\vec{1}^T = \vec{0}$$

and

$$\mathbf{Y}^T\mathbf{D}\mathbf{Y} = \mathbf{Y}^T\mathbf{D}^{1/2}\mathbf{D}^{1/2}\mathbf{Y} = (\mathbf{D}^{1/2}\mathbf{Y})^T(\mathbf{D}^{1/2}\mathbf{Y}) = \tilde{\mathbf{Y}}^T\tilde{\mathbf{Y}} = \mathbf{I}$$

implying that the columns of $\tilde{\mathbf{Y}}$ are orthonormal After the change of variable, our optimization problem becomes

$$\text{argmin}_{\tilde{\mathbf{Y}}^T\tilde{\mathbf{Y}}=\mathbf{I}_t, \tilde{\mathbf{Y}}^{1/2}\mathbf{D}^{1/2}\vec{1}=\vec{0}} tr\left(\tilde{\mathbf{Y}}^T\mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2}\tilde{\mathbf{Y}}\right). \tag{6.14}$$

We can minimize this equation by making use of the eigenvalues and eigenvectors of the (symmetric) normalized graph Laplacian

$$\mathbf{L}_{sym} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}.$$

Importantly, note that $\mathbf{L}_{sym}$ is symmetric. Furthemore, it is positive semidefinite since it can be viewed as the graph Laplacian of a graph with weights $\mathbf{W}_{ij}/\sqrt{\mathbf{D}_{ii}\mathbf{D}_{jj}}$ thus subject to the identity (6.10). Thus, it is diagonalizable with orthonormal eigenvectors $\vec{v}_1, \ldots, \vec{v}_N \in \mathbb{R}^N$ and associated nonnegative eigenvalues $\lambda_1 \leq \cdots \leq \lambda_N$ which we list in **increasing order** in this case. This leads to the first important observation

1) We could use any t of these vectors as the columns of $\tilde{Y}$ and immediately satisfy the constraint $\tilde{\mathbf{Y}}^T\tilde{\mathbf{Y}} = \mathbf{I}$.

However, we have an additional constraint and the minimization to consider. Note that $\vec{1}_N$ is an eigenvector of original graph Laplacian with eigenvalue 0. Thus, $\mathbf{L}_{sym}$ also has eigenvalue 0 with associated eigenvector $\vec{v}_1 = \mathbf{D}^{1/2}\vec{1}$ since

$$\mathbf{L}_{sym}(\mathbf{D}^{1/2}\vec{1}) = \mathbf{D}^{-1/2}(\mathbf{D}^{1/2} - \mathbf{W}\mathbf{D}^{-1/2})\mathbf{D}^{1/2}\vec{1} = \mathbf{D}^{-1/2}(\mathbf{D}-\mathbf{W})\vec{1} = \mathbf{D}^{-1/2}\mathbf{L}\vec{1} = \vec{0}.$$

As a result, all other eigenvectors of $\mathbf{L}_{sym}$ must be orthogonal to $\vec{v}_1 = \mathbf{D}^{1/2}\vec{1}$.

2) This suggests that if we drop the first eigenvector associated with eigenvalue $\lambda_1 = 0$ and use t the remaining eigenvectors of $\mathbf{L}_{sym}$ as the columns of $\tilde{\mathbf{Y}}$ we will satisfy the constraint $\tilde{\mathbf{Y}}^T\mathbf{D}^{1/2}\vec{1} = \vec{0}$.

The finaly observation is that we should choose the eigenvectors to minimize the objective.

3) We make use of the eigenvalues themselves and take

$$\tilde{Y} = \begin{bmatrix}\vec{v}_2 & \dots & \vec{v}_{t+1}\end{bmatrix}$$

so that

$$tr\left(\tilde{\mathbf{Y}}^T\mathbf{L}_{sym}\tilde{\mathbf{Y}}\right) = \lambda_2 + \cdots + \lambda_{t+1}$$

is minimized.

After undoing the change of variables, we take use the rows of

$$\mathbf{Y} = \mathbf{D}^{-1/2}\tilde{\mathbf{Y}} \in \mathbb{R}^{N\times t}$$

as our t-dimensional configuration.

6.6 Hessian Eigenmaps (HLLEs)

6.6.1 Introduction

To motivate Hessian eigenmaps, let us revisit the revisit the idea of derivatives on real valued functions on the manifold. Suppose we have a manifold $\mathcal{M} \subset \mathbb{R}^d$ and function $f : \mathcal{M} \to \mathbb{R}$. Assuming that $\mathcal{M}$ has intrinsic dimension $t < d$, to each point $\vec{x} \in \mathcal{M}$, we can associate a t dimensional tangent space $T_{\vec{x}}(\mathcal{M})$ which we equip with a choice of orthonormal basis vectors. Furthermore, there is an open neighborhood U containing $\vec{x}$ with every point in the neighborhood in one to one correspondence with a point in the tangent space allowing us to conduct calculus on the manifold using approximations within the tangent space.

Theorem 6.1 (Null Space of Hessian Operator on Manifolds)**.** *Suppose $\mathcal{M} = \Psi(\Theta)$ where Θ is an open, connected subset of $\mathbb{R}^t$, and Ψ is a locally isometric embedding of Θ into $\mathbb{R}^d$. Then $\mathcal{H}(f)$ has a $t+1$ dimensional null space spanned by the constant function and a t-dimensional space of functions spanned by the original isometric coordinates.*

Suppose now that we have a function $f : \mathcal{M} \to \mathbb{R} \subset \mathbb{R}^d$ and we wish to estimate $\mathcal{H}(f)$ using observed data $\vec{x}_1, \dots, \vec{x}_N \in \mathcal{M}$. HLLEs are based around the construction of a discretized estimation of $\mathcal{H}(f)$ which we can write in the quadratic form

$$\vec{f}^T\mathbf{H}\vec{f} \text{ where } \vec{f} = (f(\vec{x}_1), \dots, f(\vec{x}_N))^T \in \mathbb{R}^N.$$

The matrix $\mathbf{H} \in \mathbb{R}^{N\times N}$ is symmetric and positive semidefinite and depends only on the observed data $\vec{x}_1, \dots, \vec{x}_N$. Its construction will be discussed in greater detail in the algorithms section below.

For now, assume that we have access to $\mathbf{H}$. Suppose that f is in the null space of $\mathcal{H}(f)$ then i) $\vec{f}^T\mathbf{H}\vec{f} \approx 0$ and ii) from Theorem **??** it must be the case that $f(\vec{x}_i) = \alpha + \beta^T\vec{z}_i$. Importantly, if we can identify the null space of the matrix $\mathbf{H}$ then we can use this to recover the original low-dimensional coordinates (up to rigid motion and coordinate rescaling).

Similar to LLEs and ISOMAP, this algorithm begins with a k-nearest neighbor search and finishes with a eigenvalue decomposition. As inputs we provide the data, the intrinsic dimension, t, of the manifold, and a specified number of nearest neighbors k. For reasons we'll discuss later, we must select $k > t(t+3)/2$. This constraint can be problematic in practice. For example, if we believe the intrinsic dimension of $\mathcal{M}$ is 100, then we need to pick $k > 5150$ meaning we need at least 5151 samples in our dataset! Such a constraint may be difficult to satisfy in practice, but for dimension reduction focused on visualization with the ambition assumption that $t = 1, 2$, or 3, $k \geq 10$ is sufficient.

6.6.1.1 Compute nearest neighbors and local coordinates

For each $\vec{x}_i$ compute its k-nearest neighbors which we'll denote $\mathcal{N}_i$. We'll use each of these neighbors to estimate the Hessian of f at $\vec{x}_i$. Next we'll estimate local coordinates of the neighbors in $\mathcal{N}_i$ by using SVD to approximate the tangent plane $T_{\vec{x}_i}(\mathcal{M})$. One important observation to recall. We expect the points in $\mathcal{N}_i$ to all be close together and centered around $\vec{x}_i$. In this neighborhood around $\vec{x}_i$ we expect the manifold $\mathcal{M}$ to look like a small patch of $\mathbb{R}^d$. As such, they should be (nearly) contained in a t-dimensional affine subspace of $\mathbb{R}^d$.

Let $\mathbf{M}_i \in \mathbb{R}^{k\times d}$ be the matrix of centered nearest neighbors of $\vec{x}_i$. Specifically, the jth row of $\mathbf{M}_i$ is $(\vec{x}_{i_j} - \vec{x}_i)^T$ where $\vec{x}_{i_j}$ is the jth nearest neighbor of $\vec{x}_i$. We apply a singular value decomposition to $\mathbf{M}_i$ giving factorization

$$\mathbf{M}_i = \mathbf{U}_i\mathbf{S}_i\mathbf{V}_i^T$$

where $\mathbf{U} \in \mathbb{R}^{k\times k}$ abd $V_i \in \mathbb{R}^{k\times k}$ have orthonormal columns and $\mathbf{S}_i \in \mathbb{R}^{k\times d}$ is diagonal. If the neighbors in $\mathcal{N}_i$ were fully contained in a t-dimensional affine subspace we expect that $\mathbf{S}_i$ has t large singular values with the remainder equal to zero. Additionally in this case, the tangent space $T_{\vec{x}_i}$ is $\vec{x}_i + \text{span}\{\vec{v}_1, \ldots, \vec{v}_t\}$ where $\vec{v}_1, \ldots, \vec{v}_t$ are the first t columns of $\mathbf{V}$.

In practice, we should only expect the first t singular values to be large with the remainder smaller but non-zero. However, we'll still use the first t columns of $\mathbf{V}$ as an (approximate) orthonormal basis for the tangent space. The first t columns of $\mathbf{U}_i\mathbf{S}_i$ give approximations of the local coordinates of the neighbors $\mathcal{N}_i$ in this neighborhood (the jth row gives the local coordinates for the jth nearest neighbor). Hereafter, we'll let $\vec{u}_j^i = (u_{j1}^i, \ldots, u_{jt}^i)^T$ denote the local coordinates of the jth nearest neighbor in the tangent space.

6.6.1.2 Estimate the Hessian

Recall from section 6.2, that we can define derivatives for a function f at $\vec{x}_i$ using the local coordinates.

Now, we can use the local gradients and Hessians to construct a Taylor approximation to $f(\vec{x}_j)$ for each point in the neighbor $\mathcal{N}_j$. In the tangent space $T_{\vec{x}_i}(\mathcal{M})$, we associate the origin with $\vec{x}_i$. Taylor expanding around the origin to second order we have the following approximation

$$\begin{aligned} f(\vec{x}_{i_j}) &\approx f(\vec{x}_i) + \left[\nabla^{tan} f(\vec{x}_i)\right]^T \cdot \vec{u}^i_j + \frac{1}{2}(\vec{u}^i_j)^T \left(H^{tan} f(\vec{x}_i)\right) \vec{u}^i_j \\ &= f(\vec{x}_i) + \sum_{\ell=1}^{t} \left[\nabla^{tan} f(\vec{x}_i)\right]_\ell \vec{u}^i_{j\ell} + \frac{1}{2}\sum_{\ell=1}^{t} \left(H^{tan} f(\vec{x}_i)\right)_{\ell\ell} (u^i_{j\ell})^2 + \sum_{\ell<s} \left(H^{tan} f(\vec{x}_i)\right)_{\ell,s} u^i_{j\ell} u^i_{js} \end{aligned}$$

To estimate the entries of the Hessian, we use quadratic regression. Build design matrix $\mathbf{X}_i \in \mathbb{R}^{k\times(1+d+d(d+1)/2)}$ including all terms up to second order of the local coordinates such that

$$\mathbf{X}_i = \begin{bmatrix} 1 & u^i_{11} & \dots u^i_{1t} & \frac{1}{2}(u^i_{11})^2 & \dots & \frac{1}{2}(u^i_{1t})^2 & \frac{\sqrt{2}}{2}u^i_{11}u^i_{12} & \frac{\sqrt{2}}{2}u^i_{11}u^i_{13} & \dots & \frac{\sqrt{2}}{2}u^i_{1,t-1}u^i_{1t} \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & u^i_{k1} & \dots u^i_{kt} & \frac{1}{2}(u^i_{k1})^2 & \dots & \frac{1}{2}(u^i_{kt})^2 & \frac{\sqrt{2}}{2}u^i_{k1}u^i_{k2} & \frac{\sqrt{2}}{2}u^i_{k1}u^i_{k3} & \dots & \frac{\sqrt{2}}{2}u^i_{k,t-1}u^i_{kt} \end{bmatrix}$$

If we let $\vec{f_i} = (f(\vec{x}_{i_1}), \dots, f(\vec{x}_{i_k}))^T$ be a vector containing the function values at the nearest neighbors of $\vec{x}_i$, then we can estimate the coefficients of the Taylor expansion using the following regression formula

$$\vec{f_i} = \mathbf{X}^i \vec{\beta}$$

where the last $t + t(t+1)/2$ terms of $\vec{\beta}_i$ correspond to the entries of the $H^{tan}(f)$ at $\vec{x}_i$. We then estimate $\vec{\beta}_i$ using the Moore-Penrose pseudoinverse

$$\vec{\beta}_i \approx (\mathbf{X}_i^T \mathbf{X})^{-1} \mathbf{X}_i^T \vec{f_i}.$$

For the pseudoinverse to be invertible, $\mathbf{X}_i$ must have at least as many rows as columns. Thus, we need $k \geq 1 + t + t(t+1)/2 = 1 + t(t+3)/2 > t(t+3)/2$.

We can drop the first $t+1$ entries which contain the intercept and the first order regression terms. Let $\vec{\beta}_{i,drop}$ denote the final $t(t+1)/2$ entries of $\vec{\beta}_1$. If we square then sum the entries of $\vec{\beta}_{i,drop}$, we now have an estimate of $\|H^{tan}(f)\|_F^2$ at $\vec{x}_i$.

One final note on this issue. The $1/2$ and $\sqrt{2}/2$ terms in $\mathbf{X}_i$ are introduced to ensure we are correctly estimating (and single counting) the diagonal entries of the Hessian while double counting its upper triangular elements. This detail is missing from the original paper [**?**]. As such, many implementation may also replicate the mistake. Be cautious when choosing a package for implementing HLLEs!

6.6.1.3 The Eigenvalue Problem

Define $\mathbf{S}_i \in \mathbb{R}^{k \times N}$ as follows

$$(\mathbf{S}_i)_{j\ell} = \begin{cases} 1 & \vec{x}_\ell \text{ is the } j\text{th nearest neighbor of } \vec{x}_i \\ 0 & \text{else.} \end{cases}$$

Using this notation, we can express $\vec{f_i}$ (the function values from the k nearest neighbors of $\vec{x}_i$) as the product of $\mathbf{S}_i$ and $\vec{f}$ (the vector containing the function value at all samples $\vec{x}_1, \ldots, \vec{x}_N$. Specifically, $\vec{f_i} = \mathbf{S}_i \vec{f}$.

Now, we will use an empirical average of our Hessian estimates at each data point to approximate the operator $\mathcal{H}(f)$ as follows

$$\begin{aligned} \mathcal{H}(f) &= \int_{\mathcal{M}} \|H^{tan}(f)\|_F^2 dm \\ &\approx \frac{1}{N} \sum_{i=1}^{N} \| \|H^{tan}\big(f\big)(\vec{x}_i)\|_F^2 \\ &\approx \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{H}_i \vec{f_i}\|^2 \end{aligned}$$

Now, we can make use of the identity $\vec{f_i} = \mathbf{S}_i \vec{f}$ to write this approximation as a quadratic function of $\vec{f}$.

$$\begin{aligned} \approx \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{H}_i \vec{f_i}\|^2 &= \frac{1}{N} \sum_{i=1}^{N} \vec{f_i}^T \mathbf{H}_i^T \mathbf{H}_i \vec{f_i} \\ &= \frac{1}{N} \sum_{i=1}^{N} \vec{f}^T \mathbf{S}_i^T \mathbf{H}_i^T \mathbf{H}_i \mathbf{S}_i \vec{f} \\ &= \vec{f}^T \left(\frac{1}{N} \sum_{i=1}^{N} \mathbf{S}_i^T \mathbf{H}_i^T \mathbf{H}_i \mathbf{S}_i \right) \vec{f} \end{aligned}$$

Letting

$$\mathbf{H} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{S}_i^T \mathbf{H}_i^T \mathbf{H}_i \mathbf{S}_i$$

our approximation becomes

$$\vec{f}^T \mathbf{H} \vec{f}.$$

Here is the final critical observation. If f is a vector in the null space of $\mathcal{H}$ then it must be in the span of the coordinate functions, *and* we would expect the approximation to be small (close to zero except for sampling variability and estimation error). As such, we can look eigenvector(s) associated with the smallest eigenvalue(s) of $\mathbf{H}$ to give approximations to the coordinate functions!

Table 6.1: Manifold learning examples and their properties

Manifold	Intrinsic Dimension	Preimage	Preimage connected?	Preimage convex?	Isometric Mar
Helix	One	Line segment	Yes	Yes	Yes
Swiss Roll	Two	Rectangle	Yes	Yes	No
Folded Washer	Two	Annulus	Yes	No	No
Rolled Washer	Two	Annulus	Yes	No	No

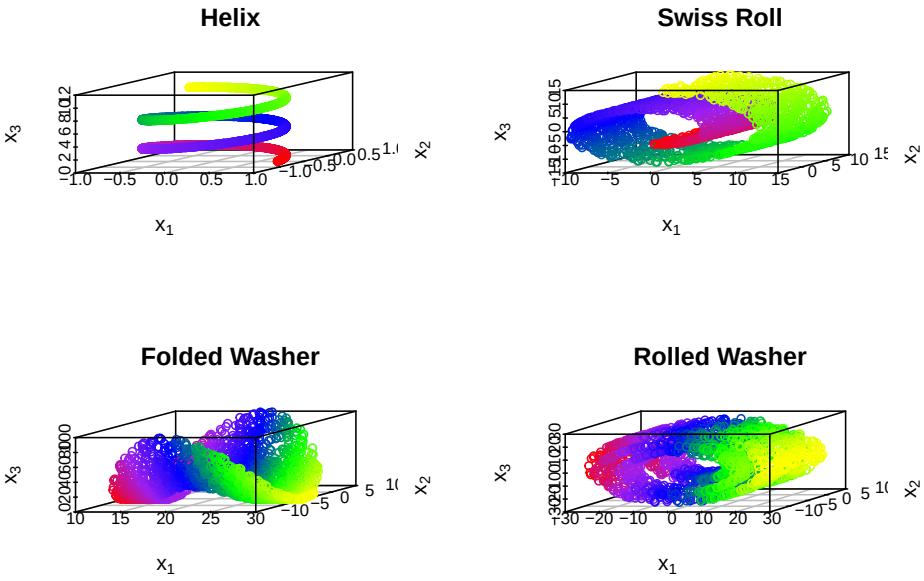
Like LLEs, $\mathbf{H}$ will have one eigenvalues which is exactly zero corresponding to the constant function. We'll take the next t eigenvectors associated with the next smallest t eigenvalues and use them to build a data matrix. The rows of this data matrix give the corresponding low dimensional coordinates of our data.

Since $\mathbf{H}$ is symmetric and positive semidefinite (its is the sum of symmetric, psd matrices), its eigenvectors will be orthogonal. Selecting these eigenvectors as approximates for functions $f_1, \ldots, f_t$ giving the corresponding 1st through tth coordinates is equiavalent to imposing an orthogonality constraint on these vectors. We'll also take the vectors to be unit length. As a result, we can expect HLLE to recover original coordinates up to rigid motion (scaling and rotation/reflection) and coordinate rescaling.

6.7 Comparison of Manifold Learning Methods

As an initial comparison, we'll look at several 3-dimensional examples with known lower dimensional latent spaces where we can visualize the data. Importantly, these examples fail to satisfy some of the conditions of the preceding methods so we can assess how susceptible the methods we have discussed to violations in their assumptions. Table x.x summarizes the results

For comparison, we provide scatterplots of 2000 samples from each manifold which we will use for analysis. The points have been color coded to improve visualization and to aid comparison with recovered lower dimensional coordinates.



6.7.1 Visual comparison of results

6.7.1.1 Helix

The helix is the simplest example and satisfies all assumptions we have discussed in the preceding sections. As such, it is not surprising that each of the methods discussed does a reasonable job compressing the data to a line segment. For the helix, we have used the isometric manifold mapping

$$\Psi(t) = \left(\frac{\sqrt{2}}{2}\cos(t), \frac{\sqrt{2}}{2}\sin t, \frac{\sqrt{2}}{2}t \right)^T$$

where $t \in [0, 15]$ is the original 1-dimensional coordinate for the data. Below we plot the recovered 1-d representation of the data against the corresponding value of t. As we have discussed, a perfect recovery of the original t is impossible. The best we can hope for is a affine relationship reflecting a one-to-one correspondence (up to translation, rescaling, and rigid motion) between the original low dimension coordinates and those recovered by our manifold learning methods.

6.7.1.2 Swiss Roll

For the Swiss roll, we use the manifold map

$$\Psi : \left[\frac{3\pi}{2}, \frac{9\pi}{2} \right] \times [0, 15] \to \mathbb{R}^3$$

given by the equation

$$\Psi(s, t) = (s\cos s, t, s\sin s)^T .$$

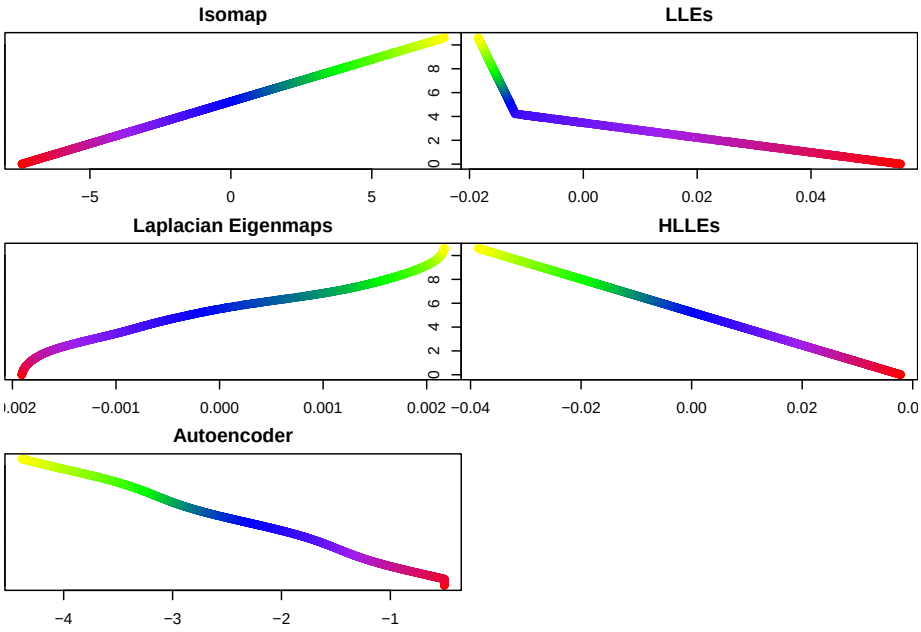


Figure 6.2: (#fig:helix_examples)Recovered latent space vs original coordinates

Below, we plot the first and second coordinates recovered by Isomap using $k = 35$ nearest neighbors.

6.7.1.3 Folded Washer

6.7.1.4 Rolled Washer

6.8 Exercises

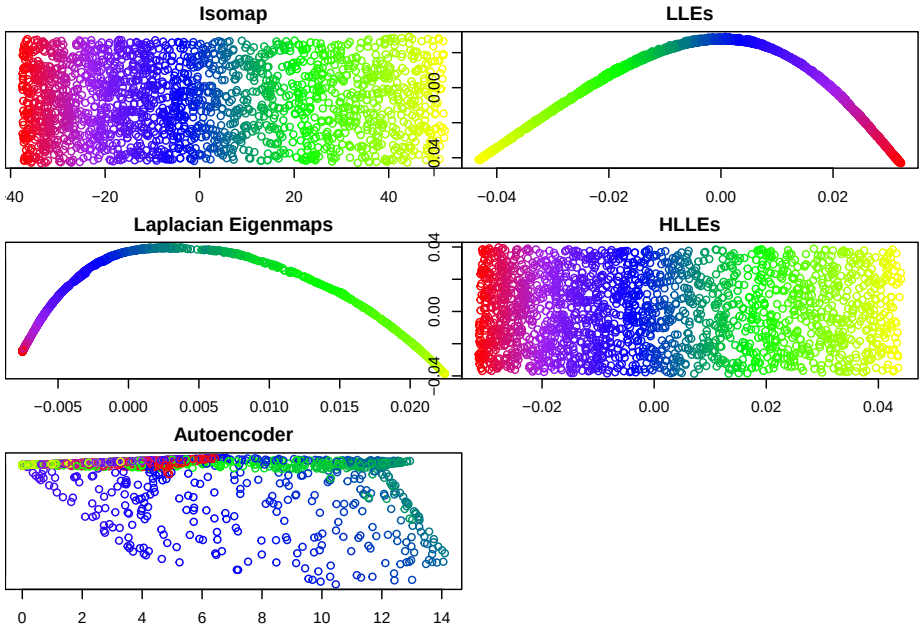


Figure 6.3: (#fig:swiss_examples)Recovered latent space vs original coordinates

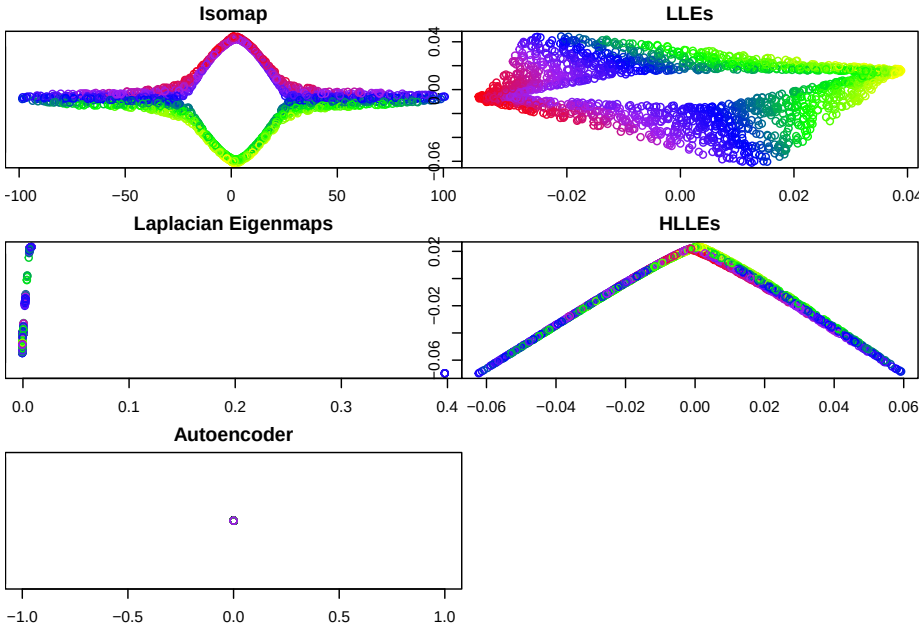


Figure 6.4: (#fig:fold_washer_examples)Recovered latent space vs original coordinates

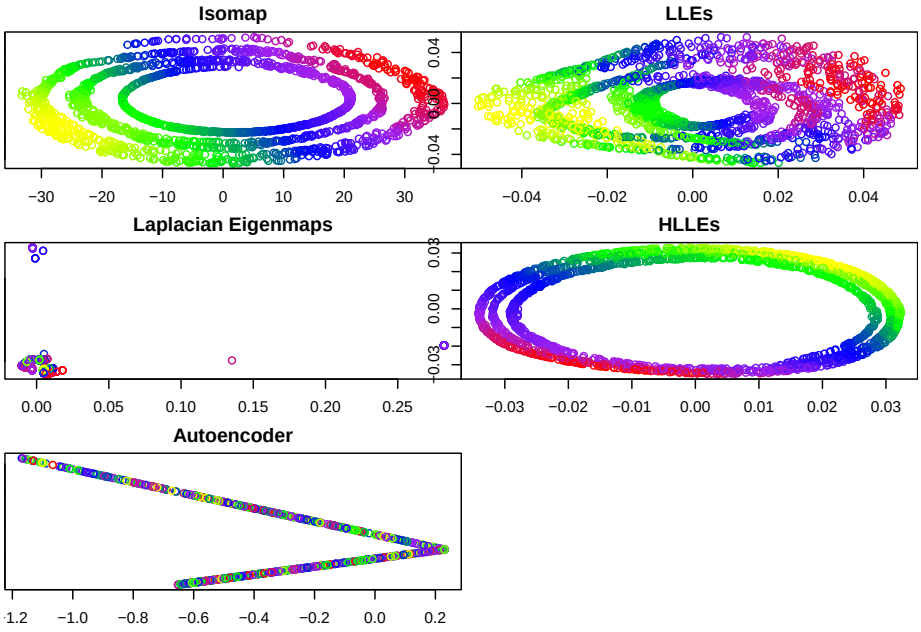


Figure 6.5: (#fig:roll_washer_examples)Recovered latent space vs original coordinates

Chapter 7

Clustering

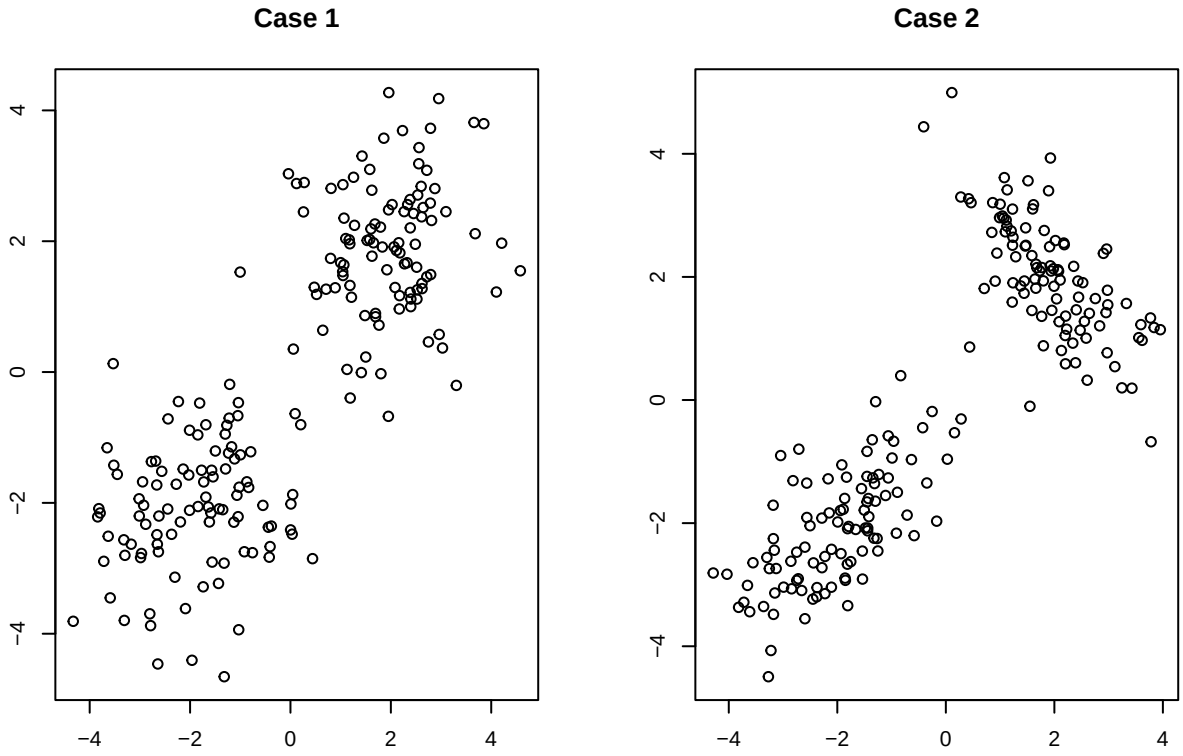
We now turn to clustering, which is the branch of UL focused on partitioning our data into subgroups of similar observations. Hereafter, we use the terms clusters and subgroups interchangeably. There are naturally a number of questions which arise including:

i) how many clusters (if any) exist?
ii) what do we mean by similar observations?

As we will see through a series of examples, these two points are linked. Given a notion of similarity (which are implicitly defined for different clustering algorithms and tuneable for others), a certain number of clusters and clustering of data may be optimal. Given a different notion of similarity, an entirely different organization of the data into a different number of clusters may arise.

For now, let us focus on the the latter question regarding similarity. Consider the following three examples of data sets in $\mathbb{R}^2$. Visually, how would you cluster the observations? In particular, how many subgroups would you say exist and how would your partition the data into these subgroups?

Example 7.1 (Different notions of similar subgroups)**.**



The questions posed above can be reasonably answered based on the figures but will not be so easy in high dimensions. We'll explore these questions and discuss data driven ways to estimate the appropriate number of cluster **for a specific algorithm** in the following sections.

There is one main commonality (and associated set of notation) that will persist throughout this section. The ultimate goal of clustering is to separate data $\vec{x}_1, \ldots, \vec{x}_N$ into k groups. The choice of k is critical in clustering. Some algorithms proceed by iteratively merging observations until only k clusters remain (hierarchical methods). Others (center- and model-based and spectral methods) treat clustering as a partitioning problem, where each element of the partition corresponds to a pre-specified number of clusters. Algorithm dependent methods for choosing k can be used for deciding optimal number of clusters without a priori knowledge.

7.1 Center-Based Clustering

Center-based clustering algorithms are typically formulated as optimization problems. Throughout this section we'll use, $C_1, \ldots, C_k \subset \{\vec{x}_1, \ldots, \vec{x}_N\}$ to denote the partitioning of our data into clusters. As such, $C_i \cap C_j$, $i \neq j$ and $C_1 \cup \cdots C_k = \{\vec{x}_1, \ldots, \vec{x}_N\}$. If $\vec{x}_i, \vec{x}_j \in C_\ell$ then we will say that $\vec{x}_i, \vec{x}_j$ are in cluster ℓ. Importantly, for each cluster we will also have an associated center

points, denoted $\vec{c}_1, \ldots, \vec{c}_k$ respectively. A point $\vec{x}_i$ is in cluster ℓ of $\vec{c}_\ell$ is the closest center to $\vec{x}_i$ under our chosen notion of distance/dissimilarity. To find a clustering of our data, we minimize a loss function dependent on the partition and/or the set of centers.

We will begin with k-means, the most well known method in this class of clustering algorithms (and perhaps all of clustering). In many ways, k-means is to clustering what PCA is to dimension reduction: a default algorithm used as a first step of analysis. Additional methods of center-based clustering include k-center and k-medoids, which we'll discuss later. Importantly, in all of these methods, the user pre-specifies a desired number of clusters, and the algorithms proceed to find an (approximately) optimal clustering. When the number of clusters in unknown, separate runs of an algorithm can be run for a range of cluster numbers. Post-processing techniques to compare different clusterings can then be used to find an 'optimal' number of clusters.

7.1.1 k-means

Given a partitioning $C_1, \ldots, C_k$, we define the intracluster variation in cluster ℓ to be

$$V(C_\ell) = \frac{1}{2|C_\ell|} \sum_{\vec{x}_i, \vec{x}_j \in C_\ell} \|\vec{x}_i - \vec{x}_j\|^2$$

be the average squared Euclidean distance between all observations in cluster ℓ. Here $|C_\ell|$ is the number of samples in cluster ℓ. If $V(C_\ell)$ is small, all points within the cluster are close together, which indicates high similarity as measured by squared Euclidean distance.

In k-means, we want all clusters to have small intracluster variation so a natural loss function is the sum of all intracluster variations

$$L_{kmeans}(C_1, \ldots, C_k) = \sum_{\ell=1}^{k} V(C_\ell).$$

With a little algebra, one can show that

$$V(C_\ell) = \sum_{\vec{x}_i \in C_\ell} \|\vec{x}_i - \vec{\mu}_\ell\|^2$$

where $\vec{\mu}_\ell = \frac{1}{|C_\ell|} \sum_{\vec{x}_i \in C_\ell}$ is the sample mean of points in cluster ℓ. Thus, the k means loss function simplifies to

$$T_{kmeans}(C_1, \ldots, C_k) = \sum_{\ell=1}^{k} \sum_{\vec{x}_i \in C_\ell} \|\vec{x}_i - \vec{\mu}_\ell\|^2.$$

Before we turn to algorithms for minimizing , let's highlight a few potential issues with k-means. Like PCA, k-means relies on squared Euclidean distance, thus it

is sensitive to outliers and imbalances in scale. These issues can be ameliorated with standardization (and potentially, the removal of outliers). However, the choice of squared Euclidean distance implicitly emphasizes a specific type of cluster shape: spheres. As will we will show with example later, k-means works well when the clusters are spherical in shape and/or far apart as measured by Euclidean distance. More complicated structure can be quite vexing for k-means.

7.1.2 k-center and k-medoids

In k-center and k-medoids, we require the cluster centers to be data points as opposed to sample means of data. Note that a given choice of centers $\vec{c}_1, \ldots, \vec{c}_k \subset \{\vec{x}_1, \ldots, \vec{x}_N\}$ implies a partition where we take

$$C_\ell = \{\vec{x}_i : \|\vec{x}_i - \vec{c}_\ell\| < \|\vec{x}_i - \vec{c}_j\|, j \neq \ell\}\,.$$

Using this fact, we can specify the loss function for k-center and k-medoids clustering.

$$L_{k-center}(\vec{c}_1, \ldots, \vec{c}_k) = \max_{\ell=1,\ldots,k} \max_{x_i \in C_\ell} \|\vec{x}_i - \vec{c}_\ell\|$$
$$L_{k-center}(\vec{c}_1, \ldots, \vec{c}_k) = \sum_{\ell=1,\ldots,k} \sum_{x_i \in C_\ell} \|\vec{x}_i - \vec{c}_\ell\|$$

In the preceding expression we have used (non-squared) Euclidean distance, which immediately makes these algorithms less sensitive to outliers and scale imbalance compared to k-means. Furthermore, alternative distance/dissimilarity measures can be used instead of Euclidean distance which adds an additional level of flexibility. In fact, we do not even need to original data in practice to implement the k-center or k-medoids. A distance matrix alone is sufficient.

Unfortunately, squared Euclidean distance has the added advantage of being much faster to implement in practice.

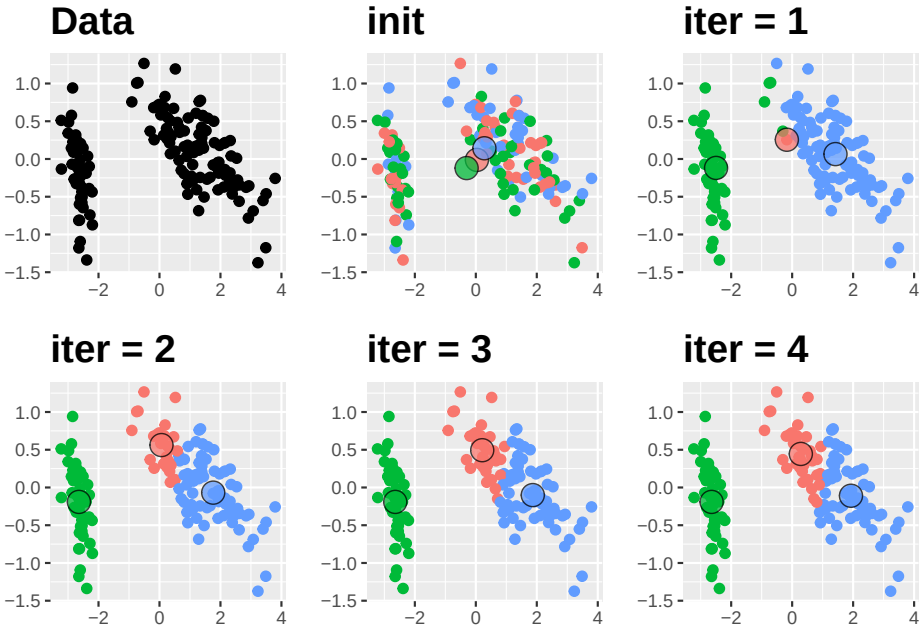
7.1.3 Minimizing clustering loss functions

For any of the three preceding methods, we want to find an optimal partitioning and/or set of centers which minimize their associated loss. One naive approach would be to search or all possible partitioning of our data into k clusters (k-means) or all possible choices of k center points from our data, but one can imagine how computationally unfeasible this approach becomes for even moderate amounts of data. Instead, we will turn to greedy, iterative algorithms which converge to (locally) optimal solutions. The standard approach for k-means is based on Lloyd's algorithm with a special initialization to avoid convergence to local minima. Later versions of this text will discuss greedy approaches for k-center and the standard partitioning around medians (PAM) algorithm for k-medoids.

7.1.3.1 Lloyd's Algorithm for k-Means

1. **Initialize**: Choose initial centers randomly from the data.
2. **Cluster Assignment**: Assign each point to the nearest center.
3. **Center Update**: Recompute cluster centers based on current assignments.
4. **Repeat** steps 2-3 until convergence.

:::{.example name="Lloyd's algorithm and Iris Data} In this example, we'll consider the three dimensional **iris** data. For visualization, we'll plot the first two PC. Centers will be outlined in black.



:::

7.1.4 Strengths and Weaknesses

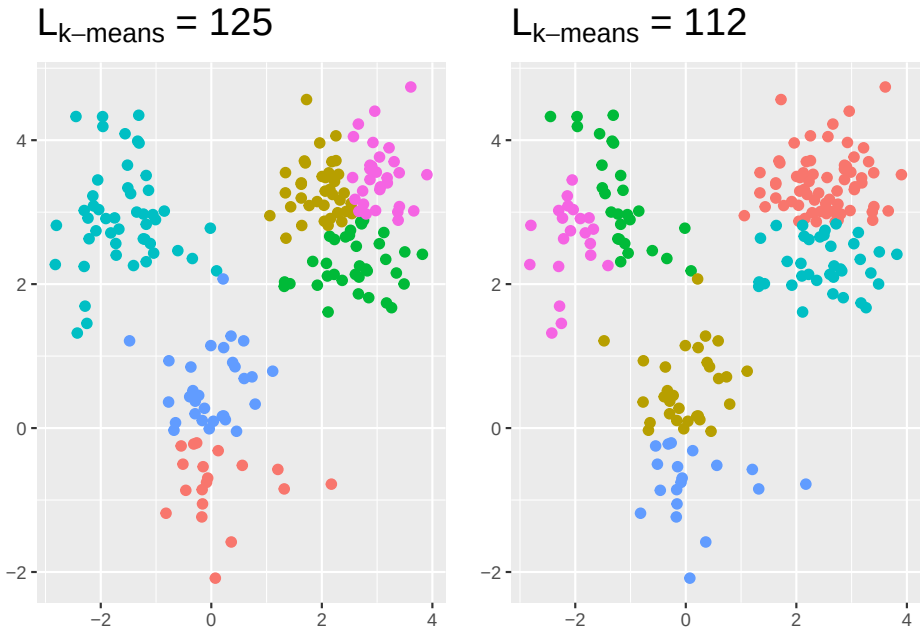
The advantages and disadvantages of center-based clustering algorithms can be separated into two main categories: computational and geometric. For now, we'll focus on the computational aspects.

In the theoretical worst case, Lloyd's algorithm can take$\propto N^{k+1}$ iterations to converge, but it is typically very fast in practice. Most implementations, allow the user to set a maximum number of iterations and converge to approximate suboptimal solutions if the maximum number of iterations is reached. PAM and greedy k-center algorithms are typically much, much slower. However, once any of the clustering algorithms are complete, new data can be merged into cluster with nearest center. Other methods we'll discuss later cannot incorporate new data without running the algorithm from scratch.

Ther are other issues that can increase the computational demands of center-

based clustering. Each of the methods requires one to choose the number of clusters in advance. We discuss this issue in more depth shortly, but for now we want to emphasize that there is no connection between the $k$cluster solution and $(k+1)$ cluster solution. Thus, when investigating a suitable number of clusters, we need to run our clustering algorithm for a range of potential values of k.

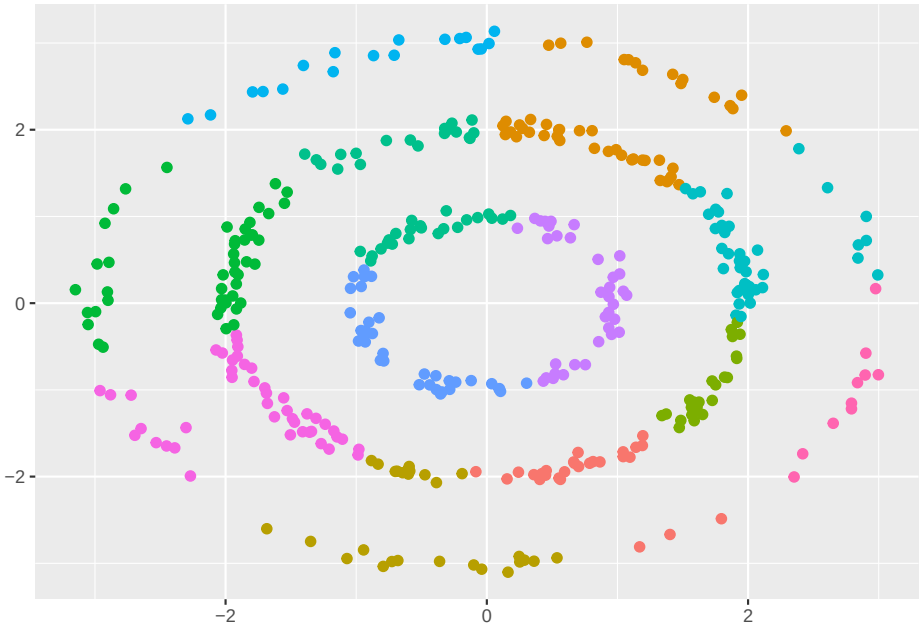
Convergence is also dependent on initial condition. In the figure below, we see two separate clustering arrangements resulting from different initial choices of the centers for k-means.



Thus, we also need to avoid convergence to local minima either by running many different initial conditions to convergence and picking the best solution or by using better initialization such as k-means++, which picks the initial points iteratively to maximize distance between the centers before running Lloyd's algorithm. Using k-means++ can reduce the number of initializations that one should use, but does not eliminate the potential for convergence to a poor clustering. Thus, center-based algorithms can become quite computationally demanding analyze appropriately.

We have already discussed the impact of scale imbalance and outliers so as we turn to the geometric aspects of center based clustering, let's first focus on the centers, which we often treat as **representative** examples of the cluster. However, the sample averages in k-means may not resemble actual data points. Though k-means may be much faster in prqctice, the centers in k-center and k-medoids are restricted to observations making them more representative of the data.

Clusters are based off interpretable notion of (Euclidean) distance so that points in a cluster are closer to one another than to points in other clusters. However, this feature biases center-based methods to clusters which are spherical in shape. If (dis)similarity not best captured through (squared) Euclidean distance, poor clustering is inevitable, and many of the methods used to estimate the number of cluster with tend to overestiamte the true value. As an example, consider the three rings shown at the beginning of this chapter. Silhouette scores (see 6.1.) suggest 12 clusters which is much larger than the three rings we see in the figure.



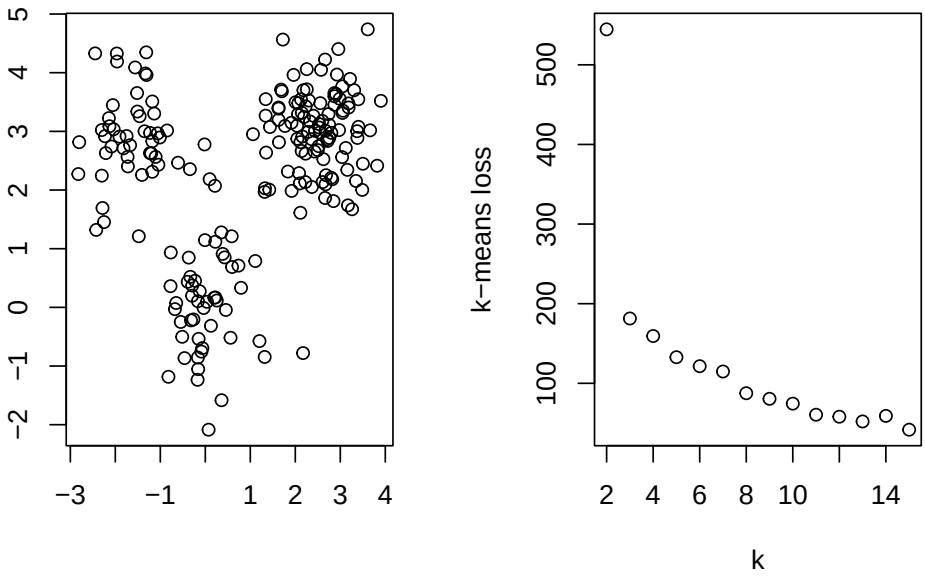
7.1.5 Choosing the number of clusters

There are many different methods which tend to give (slightly) different results. Direct methods such as the Elbow plot and silhouette diagnostics, are based on balancing a score measuring goodness of fit with a minimal number of clustering. Alternatively, one can consider testing methods such as the Gap Statistic which compare the clustering performance to a null model where clustering is absent.

7.1.5.1 Elbow plot

The center-based loss function will decrease as the number of clusters increases. Thus, we can plot the optimal loss found at different numbers of clusters and look for a sudden drop, much like we did with the scree plot in PCA. Below, we show an example of this method for data with three well separated spherical clusters

```
k <- 2:15
inertia <- rep(NA,length(k))
for (j in 1:length(k)){
  inertia[j] <- kmeans(data1,k[j])$tot.withinss
}
par(mfrow = c(1,2))
plot(data1$V1,data1$V2,xlab = '', ylab = '')
plot(k,inertia, ylab="k-means loss" )
```



7.1.5.2 Gap Statistic

The Gap statistic (Tibshirani et al., (JRSS-B, 2001)), takes a specified number of clusters and compares the total within cluster variation to the expected within-cluster variation under the assumption that the data have no obvious clustering (i.e., randomly distributed). This method can be used to select an optimal number of clusters or as evidence that there is no clustering structure. Though best suited to center based methods (particularly k-means), one can apply it to the output of any clustering algorithm in practice so long as there is some quantitative measure of the quality of the clustering.

The algorithm proceeds through the following six steps.

1. Cluster the data at varying number of total clusters k. Let $L_{k-means}(k)$ be the total within-cluster sum of squared distances using k clusters.

2. Generate B reference data sets of size N, with the simulated values of variable j uniformly generated over the range of the observed variable j . Typically $B = 500$.

3. For each generated data set $b = 1, \ldots, B$ perform the clustering for each K. Compute the total within-cluster sum of squared distances $T_K^{(b)}$.

4. Compute the Gap statistic

$$Gap(K) = \bar{w} - \log(T_K), \qquad \bar{w} = \frac{1}{B}\sum_{b=1}^{B} \log(T_K^{(b)})$$

5. Compute the sample variance

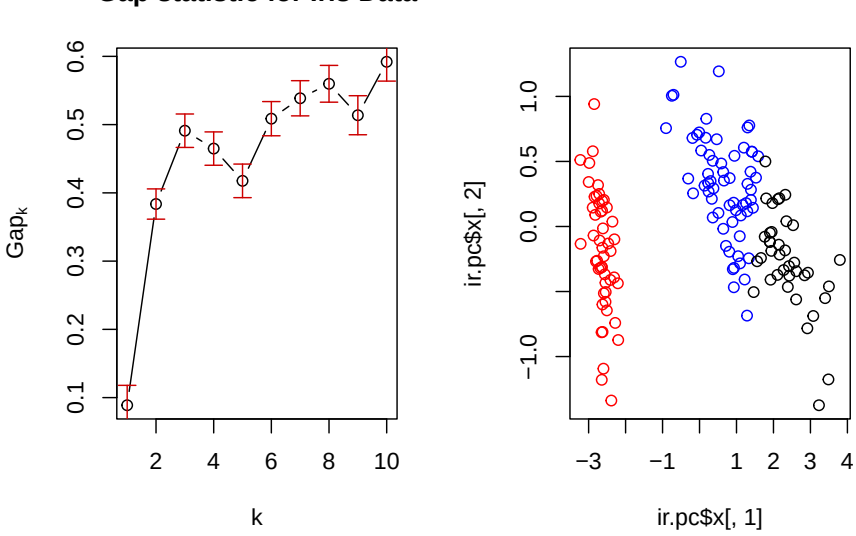
$$var(K) = \frac{1}{B-1}\sum_{b=1}^{B}\left(\log(T_K^{(b)}) - \bar{w}\right)^2,$$

and define $s_K = \sqrt{var(K)(1+1/B)}$

6. Finally, choose the number of clusters as the smallest K such that

$$Gap(K) \geq Gap(K+1) - s_{K+1}$$

::: {.example name="GAP statistic and iris data"} Below, we show a plot of the Gap Statistic (left) which indicates that three clusters is correct. The associated clustering is used to color a plot of the first two PC scores (right)



:::

7.1.5.3 Silhouette plots and coefficients

For a given clustering, we would like to determine how well each sample is clustered.

a_i = avg. dissimilarity of $\vec{x}_i$with all other samples in same cluster
b_i = avg. dissimilarity of $\vec{x}_i$ with samples in *the closest* cluster

We then define

$$s_i = \frac{b_i - a_i}{\max\{a_i, b_i\}} \in (-1, 1)$$

as the silhouette for $\vec{x}_i$.

- Observations with $s_i \approx 1$ are well clustered
- Observations with $s_i \approx 0$ are between clusters
- Observations with $s_i < 0$ are probably in wrong cluster

We can use any dissimilarity!

- Can use silhouettes as diagnostics of any method!
- A great clustering will have high silhouettes for all samples.
- To compare different values of K (and different methods), we can compute the average silhouette

  $$S_K = \frac{1}{N} \sum_{i=1}^{N} s_i$$

  over a range of values of K and choose the K which maximizes S_K.

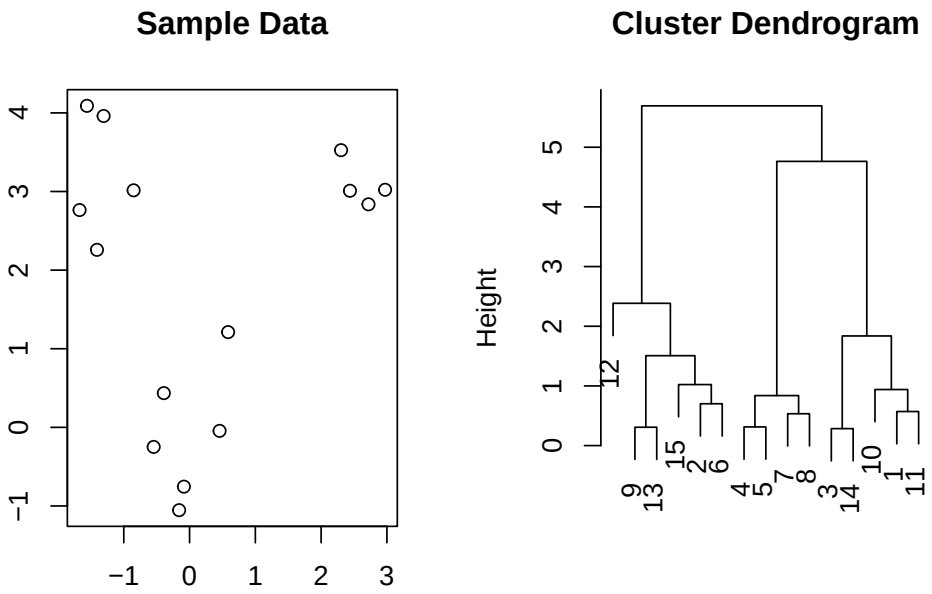
7.2 Hierarchical Clustering

Hierarchical clustering views clustering from an alternative perspective compared to the partitioning viewpoint of center-based methods. Rather than finding an optimal paritioning of data into a pre-specified number of groups, hierarchical methods treat clustering as an iterative process either merging data into larger and larger groups (agglomerative methods) or dividing the full dataset into smaller and smaller clusters (divisive methods). As a result, there are strong connections between the clustering of data into k vs $k+1$ groups. In particular, when using hierarchical methods, the optimal clustering of data into k ($k+1$) groups can be obtained by merging (splitting) the optimal clustering of data into $k+1$ (k) groups.

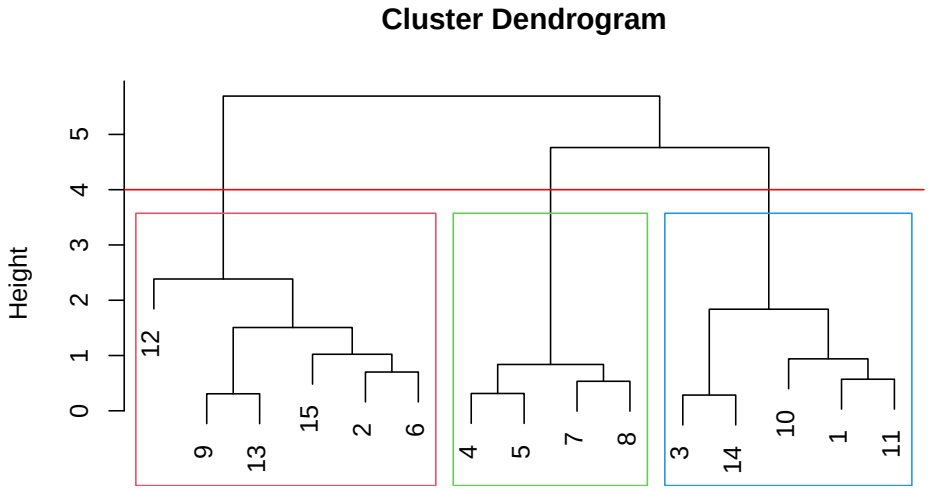
7.2.1 Dendrograms

The key output of hierarchical clustering is a dendrogram (tree diagram) depicting the subsequent merges/divisions of data, where each merge/division is shown by a horizontal line, with height indicating dissimilarity between merged or divided clusters. Before discussing the generation of a dendrogram and the many choices on which the process depends, let's first discuss how a dendrogram can be read and the information it provides for clustering.

In the figure below, samples 8 and 15 are merged into a cluster at height ≈ 0.5 indicating they have an original dissimilarity (Euclidean distance in this case) of 0.5. Sample 4 is then merged into the cluster with samples 8 and 15 at height $\approx$ indicating the dissimilarity of sample 4 with the cluster of samples 8 and 15 is ≈ 0.75



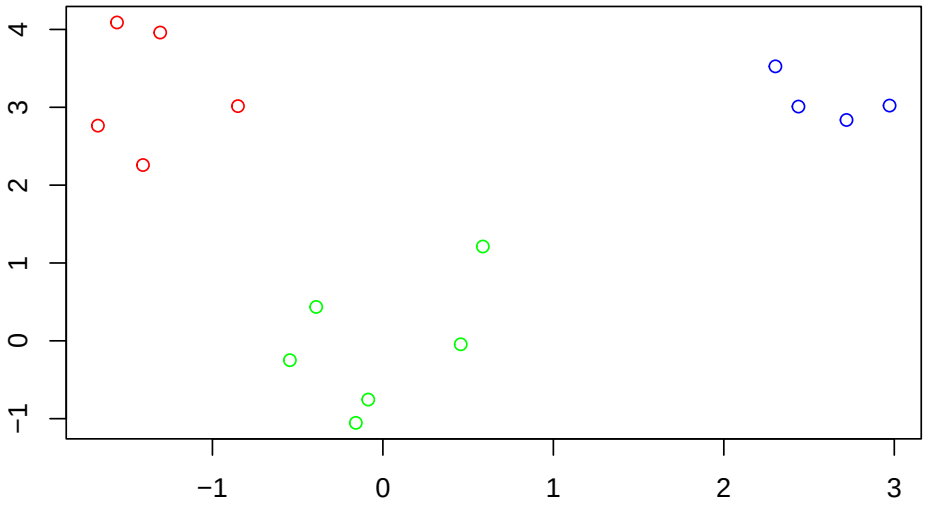
In addition to indicating the subsequent mergings/divisions of our data, the dendrogram can be used to determine a clustering of the data into a chosen number of clusters. For example, to cluster the data into three groups, we draw a horizontal line at a height (four in this case) which only cuts the tree into three branches.



Samples on the same branch are in the same cluster. Thus, the branch cut above suggest the following clustering:

```
- Cluster 1, samples: 12, 9, 13, 15, 2, 6
- Cluster 2, samples: 4, 5, 7, 8
- Cluster 3, samples: 3, 14, 10, 1, 11
```

which appears to match the clusters shown in the original data quite well.



7.2.2 Building a dendrogram

As suggested above there are two primary methods for building a dendrogram. The first is an aggolomerative approach which begins with N clusters (one per data point) then iteratively merges clusters based on the minimum dissimilarity until a final cluster containing all data remains. Alternatively, there is a divisive approach which begins with all data points in one cluster and splits iteratively until it finishes with N clusters (one per data point). Divisive clustering is less common, so we will focus on agglomerative methods hereafter.

Agglomerative clustering proceeds as follows.

1. **Initialize**: Start with `N` clusters, one per data point.
2. **Identify Closest Clusters**: Find the pair with the smallest dissimilarity.
3. **Merge Clusters**: Combine the clusters and recalculate distances.
4. **Repeat** until only one cluster remains.

Importantly, the initial dissimilarity or distance is a tuneable choice made by the practitioners. While Euclidean distance is the default, the performance of the hierarchical agglomerative clustering is highly dependent on this choice. Like k-center and k-medoids clustering, you can non-Euclidean dissimilarities or distance such as Manhattan distance or cosine (dis)similarity if they would be a better choice depending on the application.

Additionally, we have also have a choice for specifying how the pairwise dissimilarities/distances are used to compute the distance between clusters containing more than one data point. The method for computing dissimilarity/distance between clusters is called **linkage** and common methods include

- **Single Linkage**: The distance between cluster A and cluster B is the smallest distance between a sample in A and a sample in B. Using C_A and C_B to denote clusters A and B, the distance between the clusters is
$$d(C_A, C_B) = \min_{\vec{x} \in C_A, \vec{y} \in C_B} d(\vec{x}, \vec{y}).$$
- **Complete Linkage**: The largest distance between a sample in A and a sample in B is used to define the distance between clusters A and B. Mathematically, the distance between the clusters is
$$d(C_A, C_B) = \max_{\vec{x} \in C_A, \vec{y} \in C_B} d(\vec{x}, \vec{y}).$$
- **Average Linkage**: Average distance between all pairs of points in cluster A and B defines the cluster distance. Then, the distance between the clusters is
$$d(C_A, C_B) = \frac{1}{|C_A|\,|C_B|} \sum_{\vec{x} \in C_A, \vec{y} \in C_B} d(\vec{x}, \vec{y}).$$
- **Ward's Linkage**: This method uses an iterative formula based on the squared distances to minimize the within cluster variation at each merge akin to k-means. When the input to agglomerative clustering is Euclidean distance, Ward's Linkage is proportional to the squared Euclidean distance between the sample means in each cluster [**?**] though an explicit formula using the original dissimilarities is often used so that the original data is not required. For a complete discussion on this method and how it fits into the infinite family of Lance-Williams algorithms, see additional references [**??**].

In the above, the notation $d(\cdot, \cdot)$ represents the distance or dissimilarity of observations and/or clusters. Both the choice of original distances/dissimiliarities and the type of linkage have a strong impact on the quality of the clustering.

7.2.2.1 Comparing Linkage Methods

The four linkages methods above have different properties with single and complete representing extreme cases and average linkage and Ward linkage somewhere in between. Specifically, single linkage only requires two points to be close for clusters to merge, whereas complete linkage requires all pairs of points to be close. This "friend of my friend is my friend" feature of single linkage can generate clusters which are formed by long chains of singletons merged together at short heights and can in some cases capture manifold structure within a cluster. Conversely, complete linkage favors many small, compact clusters which get merged at larger heights.

Comparing and choosing a linkage method can use the above heuristics as a guide, but for a more data driven approach, Gap statistics or silhouette coefficients are suitable. Unique to hierarchical clustering, we can also use the cophenetic correlation as a quantitative measure of how well a clustering preserves pairwise distances/dissimilarities in the original data.

Definition 7.1 (Cophenetic Correlation Coefficient)**.** For compactness, let $\mathbf{\Delta}_{ij}$, $1 \leq i < j \leq N$, denote the user supplied distances/dissimilarities between observations i and j. For a given choice of linkage, let h_{ij} be the height that $\vec{x}_i$ and $\vec{x}_j$ are merged into the same cluster. The *cophenetic correlation* is the sample correlation of the $\binom{N}{2}$ pairs

$$(h_{ij}, \mathbf{\Delta}_{ij}), \qquad 1 \leq i < j \leq N.$$

The cophenetic correlation can be computed for different linkage methods. Typically, the method closest to one (optimal value of the cophenetic correlation) should be selected, though care should be used as is the case with any method to balance additional factors.

7.2.2.2 Determining the number of clusters

In addition to selecting the number of clusters, Gap statistics and silhouette coefficients can also be used to select the optimal number of clusters. Other methods, such as the Mojena coefficient [**?**] are designed specifically for hierarchical clustering.

7.3 Model-Based Clustering

Unlike the preceding methods, model-based clustering assumes a probabilistic model for the data generating process. This model implies a likelihood (or posterior if we want to take a fully Bayesian approach) which we can optimize over the model parameters. Using the optimal parameters, we can construct probability weights for cluster membership (e.g. we assign sample one to cluster one with probability 0.6 and cluster two with probability 0.4) or make a discrete decision by assigning each sample to the cluster for which it has the highest probability of membership.

For now, fix the number of clusters as$K$. Let us turn our attention to a standard two-step approach for generating a data set clustering structure which relies on two key components

1. **Cluster Densities**: Each cluster is associated with a probability density $f_j : \mathbb{R}^d \to\mapsto [0, \infty)$, for $j = 1, \ldots, K$. (In the related literature, these densities are often referred to as kernels which should be confused with the kernels we have discussed previously.)
2. **Cluster Probabilities**: Probabilities ${p_j}_{j=1}^K represent the *a priori* probability that a sample will be assigned to a given cluster

Now, suppose random vector $\vec{x} \in \mathbb{R}^d$ is generated by first choosing a cluster label Z with probability p_j, then conditional on $Z = j$ we sample x from f_j. This method implies the following joint density of $(\vec{x}, z)$

$$p(\vec{x}, Z = j) = p_j f_j(\vec{x}) \tag{7.1}$$

If we marginalize over z, we then obtain the following mixture density for $\vec{x}$,

$$p(\vec{x}) = \sum_{j=1}^{K} p_j f_j(\vec{x}) \tag{7.2}$$

We then repeat this approach independently N times to generate our data set. Hereafter, we'll use $Z_1, \ldots, Z_n \in \{1, \ldots, K\}$ to denote the **latent** cluster labels for each observation and continue using the notation $\vec{x}_1, \ldots, \vec{x}_N$ for our observed data. Under an independent sampling assumption, we then obtain the following joint distribution for the latent cluster labels and observed data

$$p(\vec{x}_1, Z_1 = z_1, \ldots, \vec{x}_N, z_N) = \prod_{i=1}^{N} p_{z_i} f_{z_i}(\vec{x}_i) \tag{7.3}$$

When we turn to likelihood estimation, a slightly different expression of the likelihood will be helpful. We can rewrite (7.3) in the following form:

$$p(\vec{x}_1, Z_1 = z_1, \ldots, \vec{x}_N, z_N) = \prod_{i=1}^{N} p_{z_i} f_j(\vec{x}_i) \tag{7.4}$$

$$= \prod_{i=1}^{N} \sum_{j=1}^{K} \mathbb{1}(z_i = j) p_j f_j(\vec{x}_i) \tag{7.5}$$

$$= \prod_{i=1}^{N} \prod_{j=1}^{K} [p_j f_j(\vec{x}_i)]^{\mathbb{1}(z_i = j)} \tag{7.6}$$

Marginalizing over $Z_1, \ldots, Z_N$ we then obtain the following joint density for the observed data

$$p(\vec{x}_1, \ldots, \vec{x}_N) = \prod_{i=1}^{N} \left(\sum_{j=1}^{K} p_j f_j(\vec{x}_i) \right). \tag{7.7}$$

As in any modeling approach, performance is strongly influenced by the flexibility of the model to capture the observed data. There are many decisions which one can use to encode beliefs of the inherent structure of the data including the number of clusters, their shape, and the proportion of samples we expect in each cluster. For remainder of this section, we will focus on Gaussian mixture models (wherein $f_1, \ldots, f_K$ are Gaussian densities) which are the most common application of model-based clustering.

7.3.1 Gaussian Mixture Models (GMM)

For a fixed number of clusters k, we assume that each cluster follows a Gaussian distribution, which are specified through their associated means and covariances. Following the notational convention from Chapter 2, we have

$$f_j(\vec{x}) = \frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}_j|} \exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu}_j)^T\mathbf{\Sigma}_j^{-1}(\vec{x}-\vec{\mu}_j)\right).$$

Following the notational convention from chapter 2, we use the shorthand

$$f_j(\vec{x}) = \mathcal{N}(\vec{x};\, \vec{\mu}_j, \mathbf{\Sigma}_j).$$

A Gaussian Mixture model (GMM) with K cluster is thus specified by the associated means, $\{\vec{\mu}\}_{j=1}^K$, covariances matrices $\{\mathbf{\Sigma}_j\}_{j=1}^K$ and probabilities $p_1, \ldots, p_K$. For brevity, we'll use

$$\Phi = \left\{\{\vec{\mu}\}_{j=1}^K, \{\mathbf{\Sigma}_j\}_{j=1}^K, \{p_j\}_{j=1}^K\right\}$$

to denote the set of model parameters. We'll use $p(\vec{x}_1, z_1, \ldots, vecx_N, z_N \mid \Phi)$ to denote the joint distribution of the latent cluster labels and observations for a given set of parameters Φ. Similarly, we'll use $p(\vec{x}_1, \ldots, \vec{x}_N \mid \Phi)$ to denote the marginal density of the observations given parameters Φ. Later, we'll consider fixing $\vec{x}_1, \ldots, \vec{x}_N$ and treating $p(\vec{x}_1, \ldots, \vec{x}_N \mid \Phi)$ as a likelihood for Φ. First, let's visualize data that can be generated by a GMM to demonstrate what shapes of clusters are possible for GMMs.

Example 7.2 (Samples from a GMM)**.** Briefly, we'll demonstrate the type of data one can generate from a Gaussian mixture model. We focus on a two-dimensional case with $K = 3$ clusters. For the sampling process, we'll use means

$$\vec{\mu}_1 = \begin{bmatrix}6\\0\end{bmatrix}, \vec{\mu}_2 = \begin{bmatrix}0\\3\end{bmatrix}, \vec{\mu}_3 = \begin{bmatrix}-3\\0\end{bmatrix},$$

and covariances

$$\mathbf{\Sigma}_1 = \begin{bmatrix}1 & 0.5\\0.5 & 1\end{bmatrix}, \mathbf{\Sigma}_2 = \begin{bmatrix}0.5 & 0\\0 & 0.5\end{bmatrix}, \mathbf{\Sigma}_3 = \begin{bmatrix}5 & -3\\-3 & 2\end{bmatrix},$$

for the Gaussian densities. And we'll use probabilities $\vec{p} = (0.5, 0.3, 0.2)$ for the cluster label assignment. Below, we show 1000 samples from this GMM with points color-coded by sampled cluster label z_i.

In the preceding example, the clusters had an elliptical shape which is a distinct variation from k-means clustering which presumes spherical clusters. The location and shape/orientation of each cluster is controlled by its mean vector and covariance matrix respectively. As such, GMMs can still generate spherical clusters if the eigenvalues of each covariance matrix are (approximately) equal. This is a natural consequence of the Gaussian distribution and diagonalization of the covariance matrix. In high dimensions, Gaussian distributions concentrate

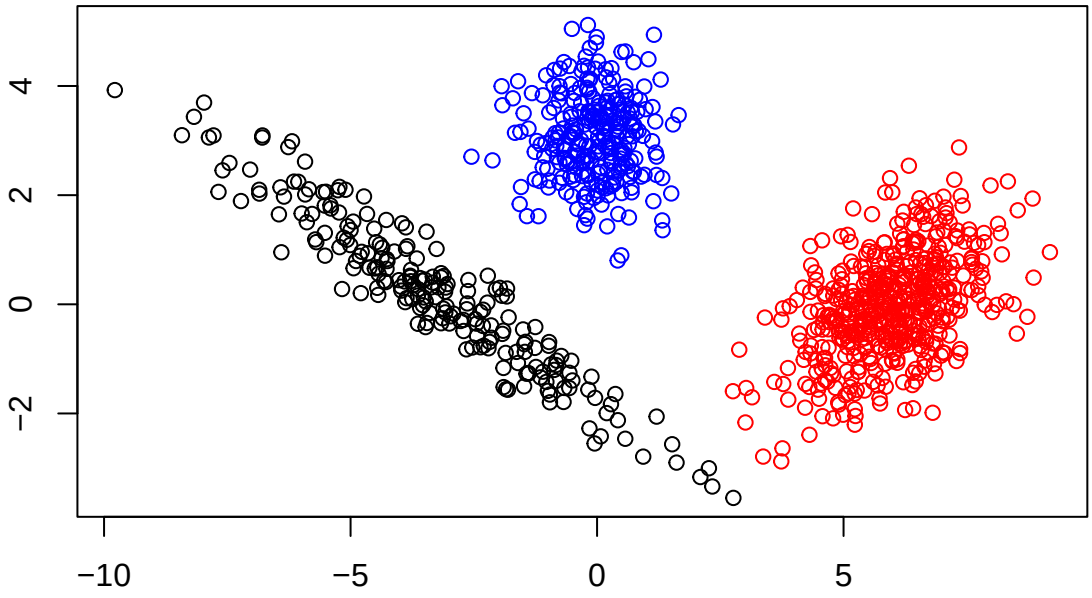
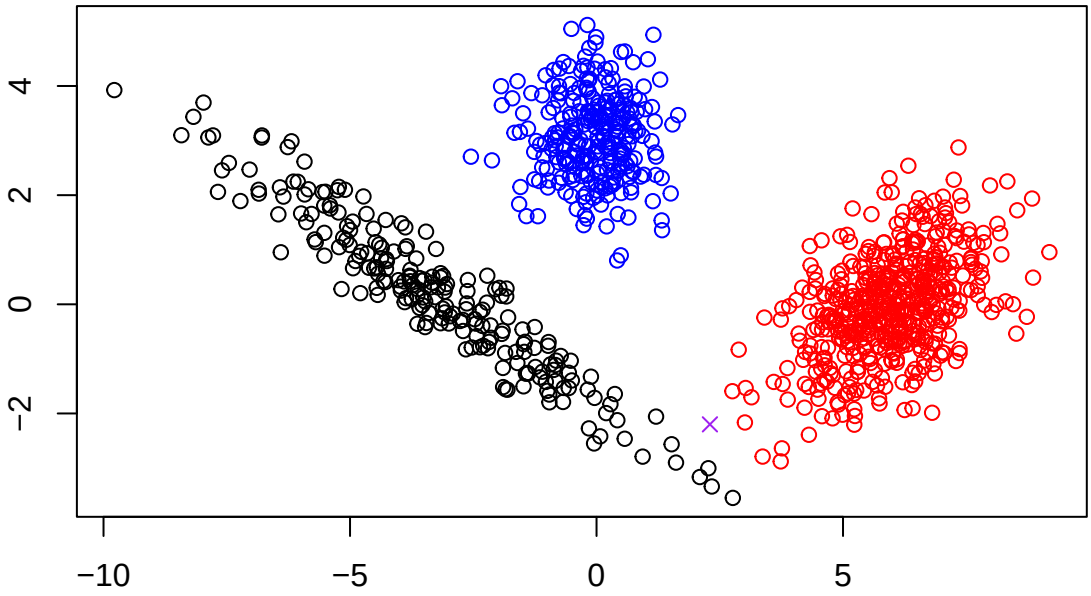


Figure 7.1: Samples from a GMM

near the surface of ellipsoids (rather than on the interior) but the general observation regarding the increased flexibility of GMMs over k-means holds true. Ignoring computational demands, one might expect GMMs to outperform k-means in many settings.

Suppose you were asked to assign a sample at the location marked with a purple x below.



Do you expect the x was generated from the same density as the points in the red cluster or from the density that generated the point in the black cluster? In center based clustering, this is a binary choice. However, it is unclear from the picture so perhaps that most honest answer is that there is a 50/50 chance that it was generated from either (we can likely agree there is vanishingly small chance it belongs to the blue cluster). The option to make such *fuzzy* clustering is a natural consequence of the probabilistic modeling approach behind GMMs. For a given parameter vector Φ we can compute a posterior $p(z_i|\vec{x}_i, \Phi)$ reflecting our belief about the clustering labeling.

Ultimately, clustering via GMMs requires estimation of the parameter Φ for a given model. If we had access to the cluster labels, $z_1, \ldots, z_N$ this would be an easy problems. We could simply take sample means and covariances for the samples assigned to cluster j to estimate $\vec{\mu}_j$ and $\mathbf{\Sigma}_j$ and we could use the proportion of samples given cluster label j as our estimate for p_j. Mathematically, we would use the estimates

$$\hat{\mu}_j = \frac{\sum_{i=1}^N \mathbb{1}(Z_i = j)\vec{x}_i}{\sum_{i=1}^N \mathbb{1}(Z_i = j)}, \hat{\mathbf{\Sigma}}_j = \frac{\sum_{i=1}^N \mathbb{1}(Z_i = j)(\vec{x}_i - \hat{\mu_j})(\vec{x}_i - \hat{\mu_j})^T}{\sum_{i=1}^N \mathbb{1}(Z_i = j)}, \hat{p}_j = \frac{\sum_{i=1}^N \mathbb{1}(z_i = j)}{N} (\#eq : simple_g \tag{7.8}$$

where $\mathbb{1}(z_i = j)$ is the indicator function taking value one if $z_i = j$ and zero otherwise.

Unfortunately, we do not have access to $z_1, \ldots, z_N$. (If we did, there would be no need to estimate the clusters.) So we need an algorithm for optimizing $p(\vec{x}_1, \ldots, \vec{x}_N \mid \Phi)$ over Φ. One approach is to use a gradient based method to

optimize the log-likelihood

$$\log p(\vec{x}_1, \ldots, \vec{x}_N \mid \Phi) = \sum_{i=1}^{N} \log \left(\sum_{j=1}^{N} p_j \mathcal{N}(\vec{x}_i;\, \vec{\mu}_j, \mathbf{\Sigma}_j) \right). \tag{7.9}$$

However, this method easily suffers from computation limitations (underflow) and additional care is required to enforce constraints on the parameters: the covariance matrices $\{\mathbf{\Sigma}_j\}_{j=1}^{K}$ must be positive definite and the probabilities $p_1, \ldots, p_K$ must be non-negative and sum to one. Instead, we'll use the powerful Expectation-Maximization (EM) algorithm which was named in the 1977 paper by Dempster, Laird, and Rubin [**?**]. In addition to proving convergence to a local maxima in our GMM setting, that paper also discusses many use cases for EM beyond mixture models.

7.3.2 Expectation-Maximization (EM) Algorithm

EM can be opaque when one is first exposed to the method. We'll discuss the mathematical details shortly. For now, let's discuss the algorithm from a high level. It begins with an initial guess for Φ. The EM algorithm is comprised of two aptly named steps.

1. **E-Step**: We use our current guess for the model parameters to estimate the cluster labels $z_1, \ldots, z_N$. These estimates are probabilistic in the sense that we compute $p(z_i \mid \vec{x}_i, \Phi)$ rather than discrete choices of the labels.
2. **M-Step**: Maximize the expected log-likelihood with respect to parameters, Φ, using the estimated label probabilities. In this step, we modify the estimates in (**??**)eq:simple_gmm_estimates) to reflect our uncertainty in the cluster labels.

The EM algorithm then proceeds by iterating the E and M steps until convergence to a local maximum is achieved. Let's explore these steps in greater detail for GMMs. Let $\Phi^{(t)}$ to denote the estimate of Φ after t iterations of the E and M steps. We'll use the superscript $^{(t)}$ above individual parameters as well.

In the E-step, we want to find $p(z_i \mid \vec{x}_i, \Phi^{(t)})$ which follows from an application of Bayes' formula

$$\begin{aligned} p(z_i = j \mid \vec{x}_i, \Phi^{(t)}) &= \frac{p(\vec{x}_i, z_i = j \mid \Phi^{(t)})}{p(\vec{x}_i \mid \Phi^{(t)})} \\ &= \frac{p(\vec{x}_i \mid z_i = j, \Phi^{(t)}) p(z_i = j \mid \Phi^{(t)})}{p(\vec{x}_i \mid \Phi^{(t)})} \\ &= \frac{p_j^{(t)} \mathcal{N}(\vec{x}_i; \vec{\mu}_j^{(t)}, \mathbf{\Sigma}_j^{(t)})}{\sum_{\ell=1}^{K} p_\ell^{(t)} \mathcal{N}(\vec{x}_i; \vec{\mu}_\ell^{(t)}, \mathbf{\Sigma}_\ell^{(t)})} \end{aligned}$$

We'll adopt the notation

$$\gamma_{ij}^{(t)} = \frac{p_j^{(t)} \mathcal{N}(\vec{x}_i; \vec{\mu}_j^{(t)}, \mathbf{\Sigma}_j^{(t)})}{\sum_{\ell=1}^{K} p_\ell^{(t)} \mathcal{N}(\vec{x}_i; \vec{\mu}_\ell^{(t)}, \mathbf{\Sigma}_\ell^{(t)})}$$

to compress our notation. Importantly, the quantities $\gamma_{ij}^{(t)}$ give a conditional distribution of $z_1, \ldots, z_N$ given $\vec{x}_1, \ldots, \vec{x}_N$ and $\Phi^{(t)}$

Now, let's turn to the handy form of the likelihood of the observations and the latent clusters labels in (7.6) which has associated log-likelihood

$$\log p(\vec{x}_1, z_1, \ldots, \vec{x}_N, z_N \mid \Phi) = \sum_{i=1}^{N} \sum_{j=1}^{K} \mathbb{1}(z_i = j) \log \left(p_j \mathcal{N}(\vec{x}_i; \vec{\mu}_j, \mathbf{\Sigma}_j) \right).$$

Treating $\vec{x}_1, \ldots, \vec{x}_N$ and Φ as fixed, we take the expectation of the above expression w.r.t. the cluster label probabilities $\gamma_{ij}^{(t)}$. From the linearity of expectation, we have

$$\begin{aligned} Q(\Phi \mid \Phi^{(t)}) &= E_{z_1,\ldots,z_N \mid \vec{x}_1,\ldots,\vec{x}_N, \Phi^{(t)}} \left[\log p(\vec{x}_1, z_1, \ldots, \vec{x}_N, z_N \mid \Phi) \right] \\ &= \sum_{i=1}^{N} \sum_{j=1}^{K} \gamma_{ij}^{(t)} \log \left(p_j \mathcal{N}(\vec{x}_i; \vec{\mu}_j, \mathbf{\Sigma}_j) \right) \\ &= \sum_{i=1}^{N} \sum_{j=1}^{K} \gamma_{ij}^{(t)} \left[\log p_j - \frac{d}{2} \log(2\pi) - \frac{1}{2} \log |\mathbf{\Sigma}_j| - \frac{1}{2} (\vec{x}_i - \vec{\mu}_j)^T \mathbf{\Sigma}_j^{-1} (\vec{x}_i - \vec{\mu}_j) \right] \end{aligned}$$

The indicators $\mathbb{1}(z_i = j)$ are replaced with the quantities $\gamma_{ij}^{(t)}$ when we take the expectation! Once we have computed $Q(\Phi \mid \Phi^{(t)})$ the E-step is complete.

For the M-step, we then optimize $Q(\Phi \mid \Phi^{(t)})$ over Φ so that

$$\Phi^{(t+1)} = \text{argmax}_\Phi \, Q(\Phi \mid \Phi^{(t)}).$$

For a GMM, there are closed form expressions for the optimal values of parameters. One can compute them directly using partial derivatives (and enforcing the constraints). We omit the calculation here and cite the result instead

$$\begin{aligned} p_j^{(t+1)} &= \frac{\sum_{i=1}^{N} \gamma_{ij}^{(t)}}{\sum_{i=1}^{N} \sum_{\ell=1}^{K} \gamma_{i\ell}^{(t)}} = \frac{\sum_{i=1}^{N} \gamma_{ij}^{(t)}}{N} \\ \vec{\mu}_j^{(t+1)} &= \frac{\sum_{i=1}^{N} \gamma_{ij}^{(t)} \vec{x}_i}{\sum_{i=1}^{N} \gamma_{ij}^{(t)}} \\ \mathbf{\Sigma}_j^{(t+1)} &= \frac{\sum_{i=1}^{N} \gamma_{ij}^{(t)} (\vec{x}_i - \vec{\mu}_j^{(t+1)})(\vec{x}_i - \vec{\mu}_j^{(t+1)})^T}{\sum_{i=1}^{N} \gamma_{ij}^{(t)}} \end{aligned}$$

The above quantities resemble the simple estimates from @ref(eq:simple_gmm_estimates). Rather than taking average using the known labels, we take weighted estimates using the relative probability of vectors have associated cluster labels!

Once we have computed $\Phi^{(t+1)}$ when then return to the E-step and continuing iterating until convergence. Convergence of the EM algorithm to a local maximum follows directly from Jensen's inequality [**?**]. Since the algorithm is only guaranteed to converge to a local maximum, one should typically investigate multiple initial conditions then select the parameter value among all optima that attains the highest log-likelihood $p(\vec{x}_1, \ldots, \vec{x}_N \mid \Phi)$.

Given a final estimate Φ^* we can now determine a clustering of the data. Let

$$\gamma_{ij}^* = \frac{p_j^* \mathcal{N}(\vec{x}_i; \vec{\mu}_j^*, \mathbf{\Sigma}_j^*)}{\sum_{\ell=1}^K p_\ell^* \mathcal{N}(\vec{x}_i; \vec{\mu}_\ell^*, \mathbf{\Sigma}_\ell^*)}$$

denote our estimate cluster label probabilities for each observation given parameter Φ^*. Each γ_{ij}^* reflects our belief that observation i was generated by the jth cluster density. To make a discrete clustering, we then assign observation $\vec{x}_i$ to the cluster maximizing γ_{ij}^*.

7.3.3 Model Selection and Parameter Constraints

The EM-algorithm provides a method for a finding the optimal parameters and subsequent clustering for a fixed number of clusters K. If we wish to estimate the number of clusters, we can use the maximized log-likelihood as a measure of the goodness of fit to the data. Let

$$\log L_k^* = \log p(\vec{x}_1, \ldots, \vec{x}_N \mid \Phi^*)$$

denote the maximum log-likelihood assuming k clusters. Larger values of $\log L_k^*$ reflect a better fit to the observed data so it natural to expect $\log L_k^*$ to increase as k increases. To avoid overfitting, we use the Bayesian Information Criterion (BIC). Let $\mathcal{P}_k$ denote the number of parameters in a GMM for k clusters. BIC is defined as

$$BIC_k = 2 \log L_k^* - 2 \log \mathcal{P}_k$$

which balances goodness of fit with model complexity. Under this framework, we select the value of k which maximizes BIC. (Note, some sources define BIC as the negative of our formula above and select k opt to minimize. Be mindful of the convention when using pre-existing packages!)

We can use the BIC approach to investigate additional constraints on the model parameters. Recall from the 7.2, the shape of clusters is controlled by the covariance matrices. By applying addition constraints to $\mathbf{\Sigma}_1, \ldots, \mathbf{\Sigma}_K$ we can reduce the number of model parameters. The standard convention for covariance

constraints comes from Banfield and Raftery (1993) who observed that every covariance matrix can be decomposed via its eigenvalue decomposition

$$\mathbf{\Sigma}_j = \lambda_j \mathbf{W}_j \mathbf{\Lambda}_j \mathbf{W}_j$$

where λ_j is a scalar parameter controlling the volume of the cluster, $\mathbf{W}_j$ is an orthonormal matrix controlling the orientation of the cluster, and $\mathbf{\Lambda}_j$ is a diagonal matrix with diagonal entries in the interval $(0, 1]$ giving the relative lengths of the semi-major axes of the ellipsoid [**?**]. We can apply constraints on one or more of the corresponding components of the covariance matrices summarized in the table below

	Volume	Shape	Orientation
Equal	$\lambda = \lambda_1 = \cdots = \lambda_K$	$\Lambda = \Lambda_1 = \cdots = \Lambda_K$	$W = W_1 = \ldots W_K$
Variable	$\lambda_1, \ldots, \lambda_K$ can differ	$\Lambda_1, \ldots, \Lambda_K$ can differ	$W_1, \ldots W_K$ can differ
Identity	Not applicable	$\Lambda_1 = \ldots \Lambda_K = I$	$W_1 = \ldots W_K = I$

We then use triplets of Equal, Variable, and Identity as shorthand for covariance constraints. For example, a GMM with covariance constraint EVE indicates covariances with equal volume, varying shape, and equal direction so that

$$\mathbf{\Sigma}_j = \lambda \mathbf{W} \mathbf{\Lambda}_j \mathbf{W}^T.$$

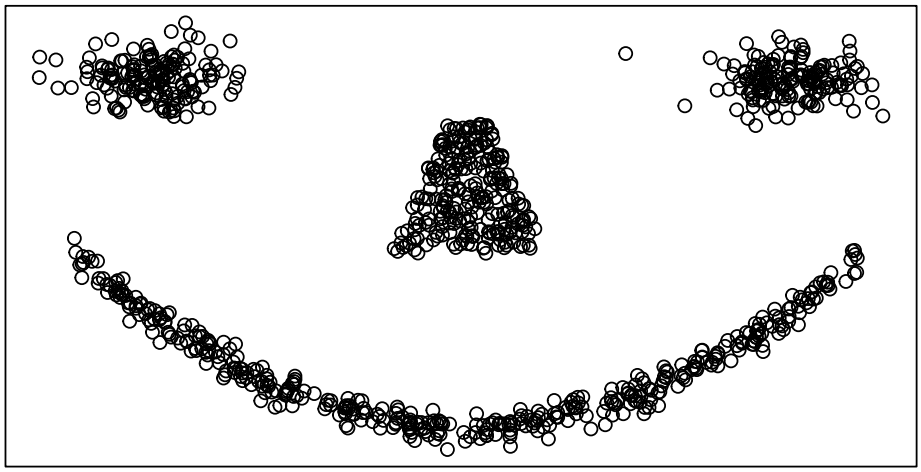
This results in a total of $2 \times 3 \times 3 - 4 = 14$ covariance options. We have subtracted 4 since some triplets coincide such as EII, EIE, and EIV.

For each covariance model, we can count the associated number of parameters and compute BIC accordingly. As a result, we have quantitative method for choosing the number of clusters and the optimal covariance model (for a given number of clusters). GMMs (with the help of the EM algorithm for fitting) provide a self-contained clustering framework with easy to interpret model selection criteria. However, there are inherent limitations which should be addressed.

7.3.4 Weakneses and Limitations

Unlike center-based methods, GMMs allow for a greater level of flexibility in the shapes of clusters (ellipses vs spheres). However, ellipsoids may still be inadequate choices a cluster supported on or near a nonlinear manifold. For example, in the **smile** data below.

Example 7.3 (GMMs fail to capture complex nonlinear structure)**.** Consider the following data in $\mathbb{R}^2$ which exhibitis four clusters upon visual inspection



The eyes and nose are each compact and approximately elliptical in shape, but the mouth is concentrated around a nonlinear parabolic curve. Applying the GMM model above for a range $K = 2, \ldots, 15$ clusters gives the following BIC curves

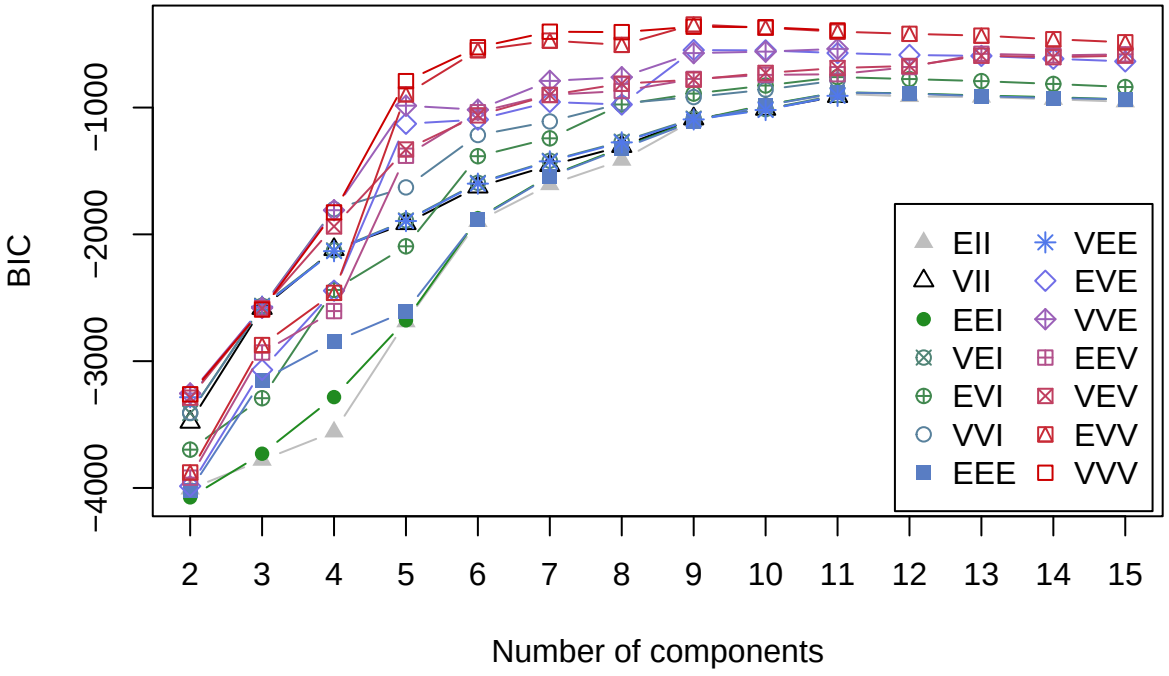
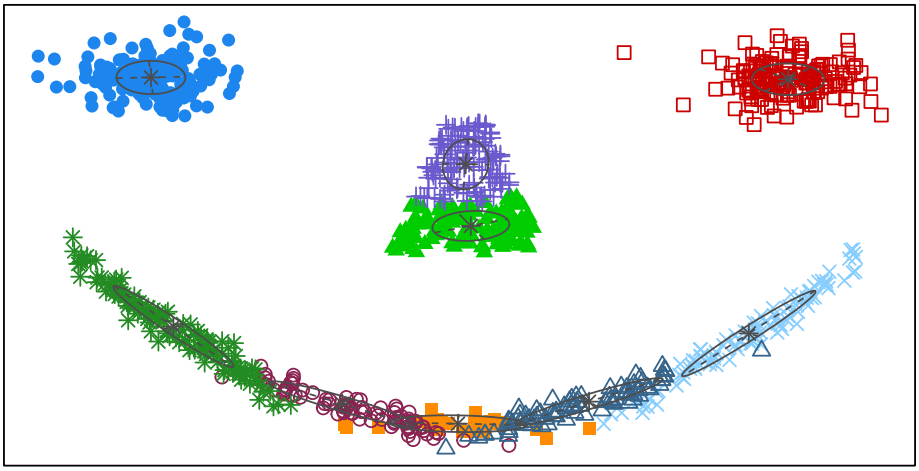


Figure 7.2: BIC curves for GMMs applied to the smile data

Using BIC, the optimal GMM one should select contains nine clusters with EVV covariance structure. In the associated clustering shown below, the eyes

are appropriately clustered, but the nose is split into two separate clusters. Most egregiously though, we can see the mouth was broken apart into five thin elliptic pieces reflecting the inherent limitations of the cluster shapes that can be identified by GMMs.

```
plot(smile.gmm, what = "classification",
     xaxt='n',yaxt='n',
     xlab = '', ylab ='')
```



Given this limitation, there has been considerable work dedicated to improving model-based clustering including the design of more expressive mixture densities [?;?;?} and methods which produce more parsimonious clustering arrangements by iteratively merging nearby clusters [?].

7.4 Spectral Clustering

7.4.1 Introduction

Spectral Clustering represents a significant leap in the evolution of clustering techniques. Distinguished from traditional methods like K-means, it excels in detecting complex structures and patterns within data. It's based on ideas from graph theory and simple math concepts, mainly focusing on how to use information from graphs. Imagine each piece of data as a point on a graph, with lines connecting the points that are similar. Spectral Clustering uses these connections to figure out how the data should be grouped, which is especially handy when the groups are twisted or oddly shaped.

The key step in Spectral Clustering is breaking down a special graph matrix (called the Laplacian matrix) to find its eigen values and eigen vectors. These eigen vectors help us see the data in a new way that makes the groups more obvious. This makes it easier to use simple grouping methods like K-means to

sort the data into clusters. This approach is great for finding hidden patterns in the data.

However, Spectral Clustering comes with its own challenges. Choosing the right number of groups can be tricky, and it might not work as well with very large sets of data because of the relatively large computing cost. Nevertheless, its robustness and adaptability have cemented its role across various domains, from image processing to bioinformatics.

7.4.2 Algorithm

1) **Similarity Graph Construction**: Start by constructing a similarity graph G from your data (similar to the steps in Laplacian Eigenmap section. Each data point is represented as a node in the graph.)

   Define the edges of the graph. There are three common ways to do this:

   i) **ϵ-neighborhood graph:** Connect all points whose pairwise distances are smaller than ϵ.

   ii) **K-nearest neighbors:** For each point, connect it to its k nearest neighbors.

   iii) **Fully connected graph:** Connect all points with each other. Typically, the Gaussian similarity function (also known as the Radial Basis Function or RBF) is used to calculate the weights of the edges: $w_{ij} = \exp(-\frac{||\vec{x}_i - \vec{x}_j||^2}{2\sigma^2})$, where $\vec{x}_i$ and $\vec{x}_j$ are two points in the dataset and σ is a tuning parameter.

Note: It is worth noting that there exists a slight difference in the construction of similarity graph matrix compared to Laplacian Eigenmap we mentioned in manifold learning chapter. For the fully connected graph, after using Radial basis to depict all the pair-wise distances, we don't need to set a threshold and sparsify the matrix (set some entries to zero) like we did in Laplacian Eigenmap, we just keep all the original radial basis distances.

2) **Graph Laplacian Matrix:** Similar to corresponding parts in Laplacian Eigenmap. Calculate the adjacency matrix W, where W_{ij} represents the weight of the edge between nodes i and j. Calculate the degree matrix D, which is a diagonal matrix where each diagonal element D_{ii} is the sum of the weights of the edges connected to node i. Compute the unnormalized Graph Laplacian matrix L as $L = D - W$.

3) **Eigen Decomposition:** Perform the eigen decomposition on the Laplacian matrix L to find its eigenvalues and eigenvectors. Select k smallest eigenvalues and their corresponding eigen vectors. k is the number of clusters you want to identify.

Mathematical Proof behind this step

As we have stated in Laplacian Eigenmap section, the Graph Laplacian matrix L is positive semi-definite.

Given any vector $\vec{y} \in \mathbb{R}^N$

$$\vec{y}^T \mathbf{L} \vec{y} = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{W}_{ij} (y_i - y_j)^2$$

Obviously, it is non-negative. Besides, we can always find a vector $\vec{y} = \mathbf{1}_N$ that makes it zero, which means the smallest eigen value must be zero, with the corresponding eigen vector being $\mathbf{1}$.

From the above equation, we can also justify our eigen-decomposition approach in finding the number of clusters. For either ϵ-neighborhood graph or K-nearest neighbors approach, if point i and j are not connected, then $\mathbf{W}_{ij} = 0$, however, if they are connected, $\mathbf{W}_{ij} = 1 > 0$. With some careful observation, we find that as long as we set $y_i = y_j$ for $\forall \mathbf{W}_{ij} > 0$, then we are able to get zero in the above equation. Since in cluster $\Omega_1 = \{\vec{x}_p, \ldots, \vec{x}_q\}$, all the points are connected, and $\mathbf{W}_{ij} > 0 \; \forall \{\vec{x}_i, \vec{x}_j\} \in \Omega_1$, we can simply set $y_i = 1$ for $\forall i \in \Omega_1$, and $y_j = 0$ for $\forall j \notin \Omega_1$. So the eigen-vector that corresponds to cluster Ω_1 is $\vec{y}_1 = \sum \vec{e}_i \in \mathbb{R}^N, \; \forall \, i \; s.t. \vec{x}_i \in \Omega_1$, where $\vec{e}_i$ is a vector with all zero except the i^{th} entry being one.

So when we perform eigen decomposition on Graph Laplacian matrix $\mathbf{L}$: $\mathbf{L} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$. Then for $\vec{y}$ s.t. $\vec{y}^T \mathbf{L} \vec{y} = 0$, we can rewrite it as

$$\vec{y}^T \mathbf{L} \vec{y} = \vec{y}^T \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T \vec{y} = (\mathbf{\Lambda}^{1/2}\mathbf{Q}^T \vec{y})^T (\mathbf{\Lambda}^{1/2}\mathbf{Q}^T \vec{y}) = 0$$

As a result, we know $\mathbf{\Lambda}^{1/2}\mathbf{Q}\vec{y} = \vec{0}$, in other words

$$\mathbf{L}\vec{y} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T \vec{y} = (\mathbf{Q}\mathbf{\Lambda}^{1/2})(\mathbf{\Lambda}^{1/2}\mathbf{Q}^T \vec{y}) = \vec{0}$$

So we know that $\vec{y}$ is just an eigen-vector of $\mathbf{L}$, with the corresponding eigen-value being zero.

In reality, especially when we use Fully-connected graph, we can't get k exact zero-eigenvalues with corresponding k eigenvectors. (Different clusters are not necessarily completely separate, and Fully-connected graph even allows every $\mathbf{W}_{ij} > 0$). So we will just conduct eigen decomposition and choose k smallest eigenvalues together with their corresponding eigen-vectors.

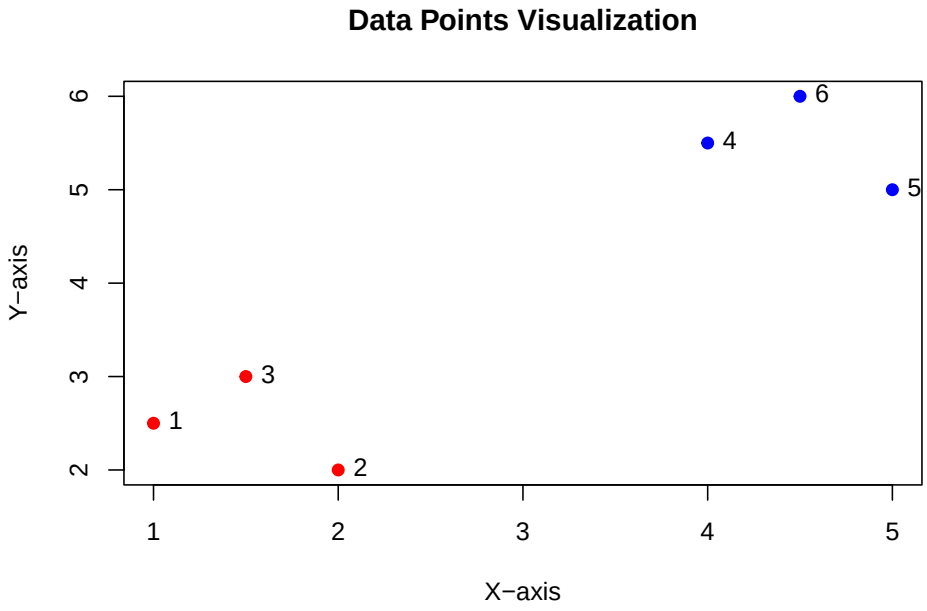
A toy example

First, we'll create the simulation data with two distinct clusters.

```
# Generate two clusters
cluster1 <- matrix(c(1, 2, 1.5, 2.5, 2, 3), ncol = 2)
cluster2 <- matrix(c(4, 5, 4.5, 5.5, 5, 6), ncol = 2)
data <- rbind(cluster1, cluster2)
```

Visualize the data points to make it more intuitive.

```
plot(data, col = c(rep("red", nrow(cluster1)), rep("blue", nrow(cluster2))), pch = 19, xlab = "X-
text(data, labels = 1:nrow(data), pos = 4, col = "black")  # Adding labels
title("Data Points Visualization")
```



We use the ϵ-**neighborhood** approach to construct the similarity graph.

```
epsilon <- 1.5  # Set epsilon value
n <- nrow(data)
similarity_matrix <- matrix(0, n, n)

for (i in 1:n) {
  for (j in 1:n) {
    if (i != j && dist(rbind(data[i, ], data[j, ])) < epsilon) {
      similarity_matrix[i, j] <- 1
      similarity_matrix[j, i] <- 1
    }
  }
}

print(similarity_matrix)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    1    1    0    0    0
## [2,]    1    0    1    0    0    0
## [3,]    1    1    0    0    0    0
## [4,]    0    0    0    0    1    1
```

```
## [5,]    0    0    0    1    0    1
## [6,]    0    0    0    1    1    0
```

Compute the Laplacian matrix.

```
degree_matrix <- diag(apply(similarity_matrix, 1, sum))
laplacian_matrix <- degree_matrix - similarity_matrix
print(laplacian_matrix)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    2   -1   -1    0    0    0
## [2,]   -1    2   -1    0    0    0
## [3,]   -1   -1    2    0    0    0
## [4,]    0    0    0    2   -1   -1
## [5,]    0    0    0   -1    2   -1
## [6,]    0    0    0   -1   -1    2
```

Perform eigen decomposition on the Laplacian matrix. We choose the smallest two eigenvalues here since we want to

```
eigen_result <- eigen(laplacian_matrix)

# Sort eigenvalues and their corresponding eigenvectors
sorted_indices <- order(eigen_result$values)
sorted_eigenvalues <- eigen_result$values[sorted_indices]
sorted_eigenvectors <- eigen_result$vectors[, sorted_indices]

# Select the smallest two eigenvalues and their corresponding eigenvectors
smallest_eigenvalues <- sorted_eigenvalues[1:2]
smallest_eigenvectors <- sorted_eigenvectors[, 1:2]
print(smallest_eigenvalues)
```

```
## [1] 1.776357e-15 1.776357e-15
```

```
print(smallest_eigenvectors)
```

```
##            [,1]       [,2]
## [1,]  0.0000000 -0.5773503
## [2,]  0.0000000 -0.5773503
## [3,]  0.0000000 -0.5773503
## [4,] -0.5773503  0.0000000
## [5,] -0.5773503  0.0000000
## [6,] -0.5773503  0.0000000
```

We find that the smallest two eigen-values are 0 (not exact zero here because of computational precision issue), and their corresponding eigen-vectors give us information about the clustering. The first cluster contains data point 4, 5, 6; while the second cluster contains the rest three data points 1, 2, 3.

We try **fully-connected graph** with radial basis function to construct the

adjacency matrix this time.

```
gamma <- 1  # Scale parameter for the RBF kernel
n <- nrow(data)
similarity_matrix <- matrix(0, n, n)

for (i in 1:n) {
  for (j in 1:n) {
    if (i != j) {
      distance <- dist(rbind(data[i, ], data[j, ]))^2
      similarity_matrix[i, j] <- exp(-gamma * distance)
    }
  }
}

print(similarity_matrix)
```

```
##              [,1]         [,2]         [,3]         [,4]         [,5]         [,6]
## [1,] 0.000000e+00 2.865048e-01 6.065307e-01 1.522998e-08 2.172440e-10 2.289735e-11
## [2,] 2.865048e-01 0.000000e+00 2.865048e-01 8.764248e-08 1.522998e-08 2.172440e-10
## [3,] 6.065307e-01 2.865048e-01 0.000000e+00 3.726653e-06 8.764248e-08 1.522998e-08
## [4,] 1.522998e-08 8.764248e-08 3.726653e-06 0.000000e+00 2.865048e-01 6.065307e-01
## [5,] 2.172440e-10 1.522998e-08 8.764248e-08 2.865048e-01 0.000000e+00 2.865048e-01
## [6,] 2.289735e-11 2.172440e-10 1.522998e-08 6.065307e-01 2.865048e-01 0.000000e+00
```

```
degree_matrix <- diag(apply(similarity_matrix, 1, sum))
laplacian_matrix <- degree_matrix - similarity_matrix
print(laplacian_matrix)
```

```
##               [,1]          [,2]          [,3]          [,4]          [,5]          [,6]
## [1,]  8.930355e-01 -2.865048e-01 -6.065307e-01 -1.522998e-08 -2.172440e-10 -2.289735e-11
## [2,] -2.865048e-01  5.730097e-01 -2.865048e-01 -8.764248e-08 -1.522998e-08 -2.172440e-10
## [3,] -6.065307e-01 -2.865048e-01  8.930393e-01 -3.726653e-06 -8.764248e-08 -1.522998e-08
## [4,] -1.522998e-08 -8.764248e-08 -3.726653e-06  8.930393e-01 -2.865048e-01 -6.065307e-01
## [5,] -2.172440e-10 -1.522998e-08 -8.764248e-08 -2.865048e-01  5.730097e-01 -2.865048e-01
## [6,] -2.289735e-11 -2.172440e-10 -1.522998e-08 -6.065307e-01 -2.865048e-01  8.930355e-01
```

```
eigen_result <- eigen(laplacian_matrix)

# Sort eigenvalues and their corresponding eigenvectors
sorted_indices <- order(eigen_result$values)
sorted_eigenvalues <- eigen_result$values[sorted_indices]
sorted_eigenvectors <- eigen_result$vectors[, sorted_indices]

# Select the smallest two eigenvalues and their corresponding eigenvectors
smallest_eigenvalues <- sorted_eigenvalues[1:2]
smallest_eigenvectors <- sorted_eigenvectors[, 1:2]
```

```
print(smallest_eigenvalues)
```

```
## [1] 2.564067e-16 2.632047e-06
```

```
print(smallest_eigenvectors)
```

```
##            [,1]       [,2]
## [1,] -0.4082483  0.4082488
## [2,] -0.4082483  0.4082494
## [3,] -0.4082483  0.4082467
## [4,] -0.4082483 -0.4082467
## [5,] -0.4082483 -0.4082494
## [6,] -0.4082483 -0.4082488
```

As we can see, this time the two smallest eigenvalues are not both zero, with one being a little bit more than zero. This has something to do with the properties of the new adjacency matrix $\mathbf{A}$. In addition, we observe that the the two eigen-vectors are not something we expected. It seems weird at the first glance, but since $\mathbf{L}\vec{y} = \vec{0}$, $\vec{y} = \vec{y}_2 \pm \vec{y}_1$ is also an eigen-vector with the eigen-value being zero, we are still able to recover the two clusters. This suggests us that we may do some computation ourselves after getting the eigen-vectors to recover the clustering situation.

In this situation, it may seem that fully-connected graph is not as straightforward as the other two adjacency matrix construction methods, and the result is also not as optimal. But we should realize that in real-data situation, different clusters are not totally separate, as a result, a soft-threshold can be a better choice in most situations.