Chapter 2: Subspace Learning

As mentioned in chapter one, datasets are present in matrix forms, in which a row is a sample or separate entity described by values (elements of a vector) given in the columns representing the different features. Each feature is a dimension to this dataset with a random variable in an acceptable domain of values with a minimum, maximum, mean, median and standard deviation and variance and covariance with other dimensions. These measures explain the dynamics or the structure of the dataset of the sample being studied. Various methods explain the different levels of analysis, such as:

- Descriptive statistics of the sample. These will be described in section 2.1.
    - Analysing the variance and covariance of the features observed or measured.
- Inferential statistics methods infer new knowledge based on the dataset sample provided. This can be causal analysis or prediction of future values.
- Linear subspace learning (LSL) reduces the data dimensionality using various methods. These methods are divided into projective methods and manifold modelling methods.
    - The projective methods identify latent variables or factors or indicators that are unobserved in the dataset but explain the correlations between the observed variables. This is useful for feature extraction and to reduce the dimensionality of a dataset. These methods use linear projections such as principal component analysis (PCA), Singular Value Decomposition (SVD), independent component analysis (ICA), linear discriminant analysis (LDA), canonical correlation analysis (CCA) and partial least squares (PLS), Factor Analysis (FA), Non-Negative Matrix Factorisation (NMF), and the generalised Nyström method. These will be described in section 2.2.
    - The manifold modelling methods learn a lower-dimensional manifold containing that data points that are embedded in the original high-dimensional space preserving the non-linear structures ignored in the projective methods. Multidimensional scaling (MDS) maps the original dataset to a lower-dimensional manifold without modelling the manifold. Methods that model the lower-dimensional manifolds include

Isometric Feature map (Isomap), locally linear embedding, and spectral clustering. These will be described in section 2.3.

- Sometimes the data are not separable in the lower dimension, and mapping to a higher dimension makes it linearly separable there. The SVM and the Kernel trick will be described in section 2.4, and the Kernel trick will be further explained using representation theory in chapter five.
- Multimodal and heterogenous datasets analysis (multiway). This process is called multilinear subspace learning (MSL), which **directly** extracts features from their multidimensional space to lower-dimensional space. These will be discussed in chapters three and onward.

2.1 Descriptive statistics and analytic statistics

Descriptive statistics explain the values of features captured in 2-dimensional arrays. A complete review of statistical analysis with python is found in books such as (Farrell, 2020). This book will review only the statistical concepts that we will need in the remaining discussions in this or the following chapters. The simplest understanding of a dataset's numerical values domains is performed by identifying the minimum, the maximum, the mean $\bar{x} = \frac{1}{N}\sum_{i=o}^{n-1} x_i$, the median = $x_{\frac{N}{2}}$, the mode (the most occurring value), and the first, second and third quartile. To identify how the values are distributed around the mean and towards the extreme values, a standard deviation is calculated as the square root of the average of the squared difference of each value of x and the mean $\overline{x}, std\ dev\ (x) = \sqrt{\frac{\sum_{i=0}^{n-1}(x_i - \bar{x})^2}{n-1}}$. The variance is the standard deviation squared, which is a more stable metric. The mean and the standard deviation are the main metrics of the central limit theorem illustrated in the normal (Gaussian) distribution. This is considered first-order statistics.

On the other side, an analysis of the observed features' variance can explain the dataset's structure and how the observed features change together and affect each other. The variance of a variable is defined as the squared standard deviation. The covariance between two features in the dataset explains how a change in one value can or can not affect the other. A covariance between two features x and y vectors are measured as $\sum = cov\ (x, y) = \frac{\sum_{i=0}^{n-1}(x_i - \bar{x})(y_i - \bar{y})}{n-1}$. A covariance square matrix is measured for all covariances between all features and each other, in which the diagonal values are the variance of one feature with its mean, and it is calculated as follows:

$$\begin{bmatrix} var(x_0) & \cdots & cov\,(x_{n-1}, x_0) \\ \vdots & \ddots & \vdots \\ cov\,(x_0, x_{n-1}) & \cdots & var(x_{n-1}) \end{bmatrix}$$

A correlation coefficient r between two features x and y is measured as $r = \frac{cov(x,y)}{\sqrt{var(x)var(y)}}$

The correlation coefficient r ranges from -1, indicating negative correlation (inverse proportional features, an increase in one causes a decrease in the other), to 0, indicating uncorrelated features, to 1, indicating positive correlation (proportional features, an increase in one causes an increase in the other). The outcome is identifying redundant features that do not add much information by being highly correlated with another feature in the dataset. This information is captured by linear Algebra as second-order statistics. The remaining chapters in the book will handle the cases of multilinear algebra, which is considered higher-order statistics.

Methods of dimensionality reductions are applied to reduce the number of features/variables to include only the features that contribute more to the variance. The above-explained methods handle the observed features as measured without the required extra preprocessing steps of handling missing values, sample size, unbalanced datasets, power analysis, and others to complete an efficient statistical analysis of a dataset.

2.2 Projective Methods - Linear Subspace Learning (LSL)

We have learned in chapter one that vectors usually are members of a vector space under linear transformations of addition and scalar multiplication. Learning a low-dimensional subspace representing a dataset is the primary method in dimensionality reduction that does not lose much information. When we multiply a vector v in a higher dimension m by a linear transformation matrix M of rows equals the high dimensions of the input vector m, and the required lower dimension in columns as n, this process projects the input vector to the lower dimension u specified by the projection matrix. $vM = u$, where $v \in \mathbb{R}^m, M \in \mathbb{R}^{m \times n}, u \in \mathbb{R}^n$.

Linear Subspace Learning (LSL) methods identify various structures in matrices using various approaches. These methods help identify the dynamics of a dataset such that the complexity can be reduced by removing linearly dependent features and redundant features and applying projection on lower dimensions capturing as much as possible from the identified structure. The simplest LSL is the rank-1 decomposition of a given matrix X of

linearly dependent columns, such that X=ab^T, where a and b are two vectors that can construct matrix X as follows:

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix} = [1 \quad 2 \quad 3] \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

This is generalised as rank-R factorisation, such that X is composed of R rank-1 components: $X = a_1 b_1^T + \cdots + a_R b_R^T = AB^T$. This will be further explained in the Singular Value Decomposition (SVD) later as the foundation for matrix compression and completion algorithms. Example matrix factorisation algorithmic approaches include principal component analysis to reduce the dimensionality while capturing the most variance, which will be further explained below. Also, Dictionary Learning algorithms aim to estimate from the known observations X, the sparse dictionary vectors and the Mixing Matrix of these vectors constructing the observations. The estimated dictionary vectors should be as sparse as possible, so combinations of these dictionary vectors can represent a high dimensional large dataset. Similarly, Factor Analysis algorithms estimate hidden/latent factors and their effects on the observations. Example Application is extracting sources in Blind Source Separation (BSS) problem; BSS aims to learn separate data components representing an entangled dataset. Each algorithm adds additional constraints to be suitable for different problems. Estimating two unknowns from one known is an underdetermined system of equations on which various iterative algorithms are proposed.

These two-way analysis methods apply covariance analysis between the features' pairs or other pair-wise statistical analyses. We can repeat the data collection in different time slots adding a third dimension of the time series. We can also divide a column into groups such as age groups 1-3, 3-5, and so forth. This needs a three-way analysis or higher that will gradually be described in this book starting from chapter three. We can correlate different entities/objects (like students and subjects) using different variables in which different modes will describe either an object or a variable based on the analysis requirements (Lu, Plataniotis and Venetsanopoulos, 2014).

Among various LSL algorithms, principal component analysis (PCA) and linear discriminant analysis (LDA) are the two most widely used in many applications. PCA is an unsupervised algorithm that does not require labels for the training samples, while LDA is a supervised method that makes use of class-specific information (Lu, Plataniotis and Venetsanopoulos, 2011).

2.2.1 Motivational Problem

A motivational problem will illustrate how to model a real-life problem into a matrix and what the eigenpairs represent to help understand eigenvalues and eigenvectors. The following table shows how three pizza kitchens on campus, labelled by their locations as A, B, and C, historically deliver pizza to all locations. Figure 1 illustrates the transition matrix of drivers' locations on the left and the graph representing the transitions on the right.

	A	B	C
A	0.3	0.3	0.4
B	0.4	0.4	0.2
C	0.5	0.3	0.2

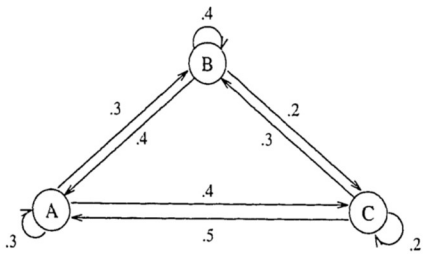


Figure 1: Pizza delivery drivers' location distribution transition matrix and graph.

The first row in this table is interpreted as follows: Kitchen in location A deliver 30% of the calls it receives to location A, 30% to location B and 40% to location C. The other two rows read the same. Delivery drivers start from the row location and end at the column location after one delivery based on the given probability distribution. The rows are the probability distribution of all possibilities and sum to 1. A driver must be at one of the three locations.

Let $p_{BB}^{(K)}$ denote the probability that a driver starts at location B, ends at B, after k deliveries

Let $p_{Ac}^{(k)}$ denote the probability that a driver starts at location A, and ends at C after k deliveries.

Same for $p_{AA}^{(k)}, p_{AB}^{(k)}, p_{Ac}^{(k)}, p_{BA}^{(k)}, p_{BB}^{(k)}, p_{Bc}^{(k)}, p_{CA}^{(k)}, p_{CB}^{(k)}$, and $p_{CC}^{(k)}$

Then predicting if a driver starts at C, what is the probability it will be at B after 2 deliveries? Based on probabilities in the first move:

$$p_{CB}^{(2)} = p_{CA}^{(1)}p_{AB}^{(2)} + p_{CB}^{(1)}p_{BB}^{(2)} + p_{CC}^{(1)}p_{CB}^{(2)} = 0.33$$

Notice that this is the dot product between row 3 (C) and column 2 (B).

Another example, if a driver starts at B, the probability of being at B after two deliveries is:

$$p_{BB}^{(2)} = p_{BA}^{(1)}p_{AB}^{(2)} + p_{BB}^{(1)}p_{BB}^{(2)} + p_{BC}^{(1)}p_{CB}^{(2)}=0.34$$

This is the dot product between row B and column B:

$$p_{CC}^{(2)} = 0.2\, p_s^{(k)} + 0.4\, p_c^{(k)} + 0.3\, p_r^{(k)}$$

$p_l^{(k)}$ is the probability vector denoting a driver's probability after k deliveries starting from location l, using the probabilities represented as a (transition) matrix S.

$$p_l^{(k)} = \begin{bmatrix} p_{lA}^{(k)} \\ p_{lB}^{(k)} \\ p_{lC}^{(k)} \end{bmatrix},\ S = \begin{bmatrix} 0.3 & 0.3 & 0.4 \\ 0.4 & 0.4 & 0.2 \\ 0.5 & 0.3 & 0.2 \end{bmatrix}$$

The transition from delivery number k to delivery k+1 is then written as the matrix-vector product (multiplication).

$$\begin{bmatrix} p_l^{(k+1)} \\ p_l^{(k+1)} \\ p_l^{(k+1)} \end{bmatrix} = \begin{bmatrix} 0.3 & 0.3 & 0.4 \\ 0.4 & 0.4 & 0.2 \\ 0.5 & 0.3 & 0.2 \end{bmatrix} \begin{bmatrix} p_l^{(k)} \\ p_l^{(k)} \\ p_l^{(k)} \end{bmatrix}$$

To know all probabilities from all locations to all locations after two deliveries, we can do matrix-matrix multiplication two times: S^2.

$$S^2 = \begin{bmatrix} 0.42 & 0.32 & 0.26 \\ 0.38 & 0.34 & 0.28 \\ 0.37 & 0.33 & 0.30 \end{bmatrix}$$

We can easily now calculate where all drivers will be after three deliveries by another matrix multiplication, tracing where they start at the rows and where they end on the columns:

$$S^2S = S^3 = \begin{bmatrix} 0.42 & 0.32 & 0.26 \\ 0.38 & 0.34 & 0.28 \\ 0.37 & 0.33 & 0.30 \end{bmatrix} \begin{bmatrix} 0.3 & 0.3 & 0.4 \\ 0.4 & 0.4 & 0.2 \\ 0.5 & 0.3 & 0.2 \end{bmatrix} = \begin{bmatrix} 0.385 & 0.333 & 0.282 \\ 0.390 & 0.334 & 0.276 \\ 0.393 & 0.333 & 0.274 \end{bmatrix}$$

Repeating this process using Python rather than by hand, we can find the probabilities for the drivers' location distribution for the next eight days:

```
import numpy as np
S = np.array(
   [[0.3, 0.3, 0.4],
   [0.4, 0.4, 0.2],
   [0.5,0.3,0.2]])
p = S
for k in range(8):
   print(p)
   p =  p.dot(S)
```

S	$S^{(1)}$	$S^{(2)}$
	$\begin{bmatrix}0.3 & 0.3 & 0.4\\0.4 & 0.4 & 0.2\\0.5 & 0.3 & 0.2\end{bmatrix}$	$\begin{bmatrix}0.41 & 0.33 & 0.26\\0.38 & 0.34 & 0.28\\0.37 & 0.33 & 0.30\end{bmatrix}$
S	$S^{(3)}$	$S^{(4)}$
	$\begin{bmatrix}0.385 & 0.333 & 0.282\\0.390 & 0.334 & 0.276\\0.393 & 0.333 & 0.274\end{bmatrix}$	$\begin{bmatrix}0.3897 & 0.3333 & 0.2770\\0.3886 & 0.3334 & 0.2780\\0.3881 & 0.3333 & 0.2786\end{bmatrix}$
S	$S^{(5)}$	$S^{(6)}$
	$\begin{bmatrix}0.38873 & 0.33333 & 0.27794\\0.38894 & 0.33334 & 0.27772\\0.38905 & 0.33333 & 0.27762\end{bmatrix}$	$\begin{bmatrix}0.388921 & 0.333333 & 0.277746\\0.388878 & 0.333334 & 0.277788\\0.388857 & 0.333333 & 0.277810\end{bmatrix}$
S	$S^{(7)}$	$S^{(8)}$
	$\begin{bmatrix}0.3888825 & 0.3333333 & 0.2777842\\0.3888910 & 0.3333334 & 0.2777756\\0.3888953 & 0.3333333 & 0.2777714\end{bmatrix}$	$\begin{bmatrix}0.38889017 & 0.33333333 & 0.2777765\\0.38888846 & 0.33333334 & 0.2777782\\0.38888761 & 0.33333333 & 0.27777906\end{bmatrix}$

Observe that the prediction starts to change less and less as we go from one delivery to the next in the future. As columns change less and less below a specified threshold, we say that the system converged.

If a vector of the initial distribution of drivers per location, such as [0.3, 0.3, 0.3], is multiplied by S, it will show the distribution of drivers after one delivery:

$$p^{(1)} = [0.3 \quad 0.3 \quad 0.3] \begin{bmatrix} 0.3 & 0.3 & 0.4 \\ 0.4 & 0.4 & 0.2 \\ 0.5 & 0.3 & 0.2 \end{bmatrix} = [0.36 \quad 0.3 \quad 0.24]$$

After two deliveries throughout the day, S^2, the right-hand side, will converge to the same number no matter what we started from.

$$p^{(2)} = [0.3 \quad 0.3 \quad 0.3] \begin{bmatrix} 0.42 & 0.32 & 0.26 \\ 0.38 & 0.34 & 0.28 \\ 0.37 & 0.33 & 0.30 \end{bmatrix} = [0.3504 \quad 0.3 \quad 0.2496]$$

After nine deliveries throughout the day, S^9, the right-hand side, will converge to the same number no matter what we started from.

$$p^{(9)} = [0.3 \quad 0.3 \quad 0.3] \begin{bmatrix} 0.38888894 & 0.33333333 & 0.27777773 \\ 0.38888887 & 0.33333333 & 0.27777779 \\ 0.38888884 & 0.33333333 & 0.27777783 \end{bmatrix}$$
$$= [0.35 \quad 0.3 \quad 0.25]$$

From the 10th delivery onward, it converges to this drivers' location vector and does not change any future: $[0.35 \quad 0.33 \quad 0.25]$

If we continue to predict the drivers' locations after many deliveries from now by executing the previous python loop for 64 iterations for the same probability distributions of calls and deliveries, we will observe that the prediction does not change very much. The resulting drivers' locations vectors will hold approximately the prediction for what typical location they might be at after any number of deliveries.

If we change the initial drivers' locations vector to reflect any initial distributions, we will also end up with the same typical drivers' locations prediction. Try the code snippet, you will find that the code converged to the same vector as before on the 10th iteration only.

At any given time step, 35% of the drivers will be at location A, 33% at location B, and 25% at location C. The drivers' initial distribution should not change the typical distribution after several deliveries.

These are called Markov Processes. These kinds of techniques apply to many problems. For example, the Google page rank algorithm determines which website is the most important (highest rank) based on the probability that the user will select the page link from several incoming links to other pages. Another example is predicting the next day's weather based on today's weather using probabilities collected over several days in a given location. We predicted the next drivers' location distribution after one delivery by multiplying the probabilities of calls location distribution and transitions in matrix S by the current drivers'

location distribution: $\mathbf{d^{(k+1)} = Sd^{(k)}}$, which can be extended to any number of deliveries prediction by $\mathbf{d^{(k)} = S^k d^{(0)}}$, to observe that eventually $\mathbf{d^{(k+1)} \approx Sd^{(k)}}$. This diminished change means that $\mathbf{d^{(k+1)}}$ came arbitrarily close to an eigenvector, x, associated with the eigenvalue $\boldsymbol{\lambda}$ =1 of matrix S: $\mathbf{Sx = \lambda x}$. For transition matrices, the dominant Eigenvalue $\boldsymbol{\lambda}$ is1.

The dominant Eigenvector represents the typical drivers' location distribution in the previous example. This is known as the (algebraic) eigenvalue problem. Scalars λ that satisfy Ax = λx for non-zero vector x are known as Eigenvalues, while the corresponding non-zero vector x to each λ Eigenvalue is known as Eigenvectors. From the previous calculations, we can answer questions like "what is the typical drivers' location distribution for this Pizza kitchen calls' distribution?" (Answer: 35% at A, 33% at B, and 25% at C). A similar approach can be used to answer questions like "what is the most requested pizza topping in a given order?"

The power method finds an eigenvector associated with the largest Eigenvalue (in magnitude). It starts by guessing an initial value for x, then loops multiplying Ax to get new x, optionally dividing the new x, by the last element of the previous value (or any normalisation step), until the new value does not change (much) anymore. The dominant Eigenvalue is the Rayleigh quotient of eigenvector x, which is $\frac{x^T Ax}{x^T x}$.

```
import numpy as np
rand=np.random.RandomState(235)
A = np.array(rand.random(size=(3, 3)))
x = np.array(rand.random(size=3))
x_Old = np.array(rand.random(size=3))
i=0
while np.isclose(x, x_Old).all() == False:
   i=i+1
   x_Old = x
   x =  A.dot(x)
   if x[-1] != 0:
      x = np.true_divide(x, x[-1])
   print("i: ", i, " x = ", x)
eigval = np.true_divide(x.T.dot(A.dot(x)), x.T.dot(x))
print ("Dominant eigen value = ", eigval)
```

Matrices, Markov chains, Eigenvalue and Eigenvectors have many real-world applications. There are many, many examples of the use of Markov chains. A brief look at some significant applications can be found in (VON HILGERS and LANGVILLE†, no date) (Geijn, 2012), (Carter, 1995).

2.2.2 Eigendecomposition

The algebraic eigenvalue problem is a matrix decomposition method that reveals various algebraic properties that make working with matrices easier. Scalars $\boldsymbol{\lambda}$ are the eigenvalues of matrix A that satisfy Ax = $\boldsymbol{\lambda}$x for non-zero eigenvectors x. The pizza delivery drivers' location distribution examples in the previous section illustrate how a Markov Process uses a transition Matrix, such that the power method converges to the most dominant Eigenvector corresponding to the largest Eigenvalue, which was 1 for transition matrices. Then we saw how to generalise the power method to identify the most dominant eigenpair for any given non-transition matrices using the Rayleigh quotient for the dominant Eigenvalue. We need another method to find all eigenvalues and corresponding eigenvectors (eigenpairs). Eigenvalues and eigenvectors often explain the bases that are invariant to linear transformations. All eigenvalues are non-negative, real numbers because covariance matrices are symmetric and positive semi-definite.

In Python:

```
# eigen decomposition of a matrix A
from numpy import linalg as LA
eignVal, eignVec = LA.eig(A)
```

2x2 Matrix Example:

To find the Eigenvalues and Eigenvectors of a matrix A $\in \mathbb{R}^{2\times2} = \begin{bmatrix} 7 & 3 \\ 3 & -1 \end{bmatrix}$, we need to perform the following steps:

Step 1: Find the characteristic polynomial of $\boldsymbol{A}$. Since $Ix = x$ for any given vector x and the identity matrix I, then Ax= $\lambda\boldsymbol{I}$x, therefore Ax- $\lambda\boldsymbol{I}$x $= 0 \rightarrow$ (A- $\lambda\boldsymbol{I}$)x $= 0$, for Eigenvector x and Eigenvalue λ. Since x should be a non-zero vector, solving for (A- $\lambda\boldsymbol{I}$) $= 0$, by calculating the determinant is called the characteristic equation. This will find the non-zero solution.

$$\boldsymbol{M}=\boldsymbol{A}-\lambda\boldsymbol{I} = \begin{bmatrix} 7-\lambda & 3 \\ 3 & -1-\lambda \end{bmatrix}$$

Det(M) $= |\boldsymbol{M}| = |\boldsymbol{A}-\lambda\boldsymbol{I}|$, using the 2x2 determinant rule:

$$= \begin{bmatrix} 7-\lambda & 3 \\ 3 & -1-\lambda \end{bmatrix}$$

$$= (7-\lambda)(-1-\lambda) - 9 = \lambda^2 - 6\lambda - 16 = (\lambda-8)(\lambda+2)$$

Step 2: Find the roots of the characteristic polynomial of $\boldsymbol{A}$ to obtain the Eigenvalues of $\boldsymbol{A}$.

$(\lambda - 8)(\lambda + 2) = 0$ **Then the roots are** $\lambda_1 = 8, \lambda_2 = -2$

Step 3: Repeat the following (i) and (ii) steps for each Eigenvalue λ of $\boldsymbol{A}$

For Eigenvalue $\lambda_1 = 8$:	For Eigenvalue $\lambda_2 = -2$:
(i)Form the matrix $\boldsymbol{M}=\boldsymbol{A}-5\boldsymbol{I}$ $\begin{bmatrix} 7-8 & 3 \\ 3 & -1-8 \end{bmatrix} = \begin{bmatrix} -1 & 3 \\ 3 & -9 \end{bmatrix}$	(i)Form the matrix $\boldsymbol{M}=\boldsymbol{A}-3\boldsymbol{I}$ $\begin{bmatrix} 7+2 & 3 \\ 3 & -1+2 \end{bmatrix} = \begin{bmatrix} 9 & 3 \\ 3 & 1 \end{bmatrix}$
(ii)Find the solution of $\boldsymbol{Mx}=\boldsymbol{0}$(These non-zero vectors are linearly independent Eigenvectors of $\boldsymbol{A}$ belonging to λ.) $\begin{bmatrix} -1 & 3 \\ 3 & -9 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0$ $-x_1 + 3x_2 = 0$, and $3x_1 - 9x_2 = 0$ Then $x_1 = 3x_2$ The Eigenvector x= $\begin{bmatrix} 3x_1 \\ x_1 \end{bmatrix}$, or $x_1\begin{bmatrix} 3 \\ 1 \end{bmatrix}$, for $x_1 \neq 0$ such that $\begin{bmatrix} 3 \\ 1 \end{bmatrix}$is one possible Eigenvector.	(ii)Find the solution of $\boldsymbol{Mx}=\boldsymbol{0}$(These non-zero vectors are linearly independent Eigenvectors of $\boldsymbol{A}$ belonging to λ.) $\begin{bmatrix} 9 & 3 \\ 3 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0$ $9x_1 + 3x_2 = 0$, and $3x_1 + x_2 = 0$ Then $3x_1 = -x_2$ The Eigenvector x=$\begin{bmatrix} x_1 \\ -3x_1 \end{bmatrix}$or $x_1\begin{bmatrix} 1 \\ -3 \end{bmatrix}$ for $x_1 \neq 0$ such that $\begin{bmatrix} 1 \\ -3 \end{bmatrix}$is one possible Eigenvector.

3x3 Matrix Example:

To find the Eigenvalues and Eigenvectors of a matrix A $\in \mathbb{R}^{3\times3}$ = $\begin{bmatrix} 4 & 1 & -1 \\ 2 & 5 & -2 \\ 1 & 1 & 2 \end{bmatrix}$, we follow the same steps as above:

Step 1: Find the characteristic polynomial of $\boldsymbol{A}$.

$$\boldsymbol{M}=\boldsymbol{A}-\lambda\boldsymbol{I} = \begin{bmatrix} 4-\lambda & 1 & -1 \\ 2 & 5-\lambda & -2 \\ 1 & 1 & 2-\lambda \end{bmatrix}$$

Det(M) $= |\boldsymbol{M}|=|\boldsymbol{A}-\lambda\boldsymbol{I}|$, using the 3x3 determinant rule:

$$= \begin{bmatrix} 4-\lambda & 1 & -1 \\ 2 & 5-\lambda & -2 \\ 1 & 1 & 2-\lambda \end{bmatrix}\begin{matrix} 4-\lambda & 1 \\ 2 & 5-\lambda \\ 1 & 1 \end{matrix}$$

$$= (4-\lambda)(5-\lambda)(2-\lambda) - 2 - 2 + (5-\lambda) + 2(4-\lambda) - 2(2-\lambda)$$

$$= -\lambda^3 + 11\lambda^2 - 39\lambda + 45 = -(\lambda-5)(\lambda-3)^2$$

Step 2: Find the roots of the characteristic polynomial of $\boldsymbol{A}$ to obtain the eigenvalues of $\boldsymbol{A}$.

$-(\lambda-5)(\lambda-3)^2=0$ **The roots are** $\lambda_1=5,\lambda_2=3$

Step 3: Repeat (i) and (ii) for each eigenvalue λ of $\boldsymbol{A}$

For Eigenvalue $\lambda_1=5$:	For Eigenvalue $\lambda_2=3$:
(i)Form the matrix $\boldsymbol{M}=\boldsymbol{A}-5\boldsymbol{I}$ $\begin{bmatrix}4-5 & 1 & -1\\ 2 & 5-5 & -2\\ 1 & 1 & 2-5\end{bmatrix}=\begin{bmatrix}-1 & 1 & -1\\ 2 & 0 & -2\\ 1 & 1 & -3\end{bmatrix}$	(i)Form the matrix $\boldsymbol{M}=\boldsymbol{A}-3\boldsymbol{I}$ $\begin{bmatrix}4-3 & 1 & -1\\ 2 & 5-3 & -2\\ 1 & 1 & 2-3\end{bmatrix}=\begin{bmatrix}1 & 1 & -1\\ 2 & 2 & -2\\ 1 & 1 & -1\end{bmatrix}$
(ii)Find the solution of $\boldsymbol{M}\boldsymbol{x}=\boldsymbol{0}$(These non-zero vectors are linearly independent Eigenvectors of $\boldsymbol{A}$ belonging to λ.) $\begin{bmatrix}-1 & 1 & -1\\ 2 & 0 & -2\\ 1 & 1 & -3\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=0$	(ii)Find the solution of $\boldsymbol{M}\boldsymbol{x}=\boldsymbol{0}$(These non-zero vectors are linearly independent Eigenvectors of $\boldsymbol{A}$ belonging to λ.) $\begin{bmatrix}1 & 1 & -1\\ 2 & 2 & -2\\ 1 & 1 & -1\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=0$
Then $x_1=x_3, x_2=2x_1$	Then $x_1+x_2=x_3$
The Eigenvector x= $\begin{bmatrix}x_1\\ 2x_1\\ x_1\end{bmatrix}$, or $x_1\begin{bmatrix}1\\ 2\\ 1\end{bmatrix}$, for $x_1\neq 0$ such that $\begin{bmatrix}1\\ 2\\ 1\end{bmatrix}$ is one possible Eigenvector.	The Eigenvector x= $\begin{bmatrix}x_1\\ x_2\\ x_1+x_2\end{bmatrix}$, or $x_1\begin{bmatrix}1\\ 0\\ 1\end{bmatrix}+x_2\begin{bmatrix}0\\ 1\\ 1\end{bmatrix}$, for $x_1\neq 0, x_2\neq 0,$ such that $\begin{bmatrix}1\\ 1\\ 2\end{bmatrix}$ is one possible Eigenvector.

PS the Product of Eigenvalues is equal to the determinant of the matrix. For the First example, det(A) = det $\begin{pmatrix}7 & 3\\ 3 & -1\end{pmatrix}=-16=8$ x -2 = -16. For the second example, det(A) = 4 . det $\begin{pmatrix}5 & -2\\ 1 & 2\end{pmatrix}$-1 . det $\begin{pmatrix}2 & -2\\ 1 & 2\end{pmatrix}$-1 . det $\begin{pmatrix}2 & 5\\ 1 & 1\end{pmatrix}=45=5\,x\,3\,x\,3=45$

Eigenvalues and Eigenvectors of a square matrix have many applications in Communication systems, engineering disciplines such as in designing bridges, art applications such as music composition, and signal processing such as designing car stereo systems and concert halls, among many more. An online calculator can be found at https://www.symbolab.com/solver/matrix-eigenvalues-calculator.

This simple method is not computationally efficient for larger matrices, as it might attempt to create memory a typical computer might not have. An algorithm based on the QR factorisation is used, and various iterative methods are similar to the power method and its convergence criteria. Calling any of these functions such as numpy.linalg.eig, usually achieve speed-ups by employing the available hardware in the machine to process the computation in parallel.

Another computational consideration is the Roundoff errors. These affect small and large matrices equally. $\frac{1}{3}$ is stored as a rounded decimal point figure using normal notation. In normal notation, a number is written as m x 10^a where $0.1 \leq |m| < 1$. (In scientific notation, $1 \leq |m| < 10$), where m is mantissa and a is abscissa. The number 1234 is represented as 0.1234 x 10^4, mantissa = 0.1234 and abscissa = 4. This representation affects the accuracy of any numerical calculations, particularly multiplication. This is why the log of any value is often used to turn multiplication into an addition operation.

2.2.3 Principal Component Analysis (PCA)

The principal components analysis is defined as linear feature projection on new orthogonal coordinates such that it captures the most variance in the first coordinate (principal component) and lesser remaining variance in decreasing order in the following coordinates. These principal components are the eigenvectors of the dataset and are orthogonal to each other and considered uncorrelated factors. Finding the projection of the original dataset onto the most significant initial principal components (typically 2 or 3) will produce an uncorrelated model that captures the most variance of the original dataset in which the signal-to-noise ratio is highest. The steps to calculate the PCA are simply calculating the eigenvectors of the covariance matrix of a dataset.

PCA can be calculated using a direct projection matrix such that $y = U^T x$, where $y \in \mathbb{R}^p$ is the projected data, $U \in \mathbb{R}^{m \times p}$ is the projection matrix containing the p Eigenvectors, and $x \in \mathbb{R}^m = (x_m - \bar{x})$ is the centred m-dimensional dataset standardised to zero-mean. The projected dataset on each principal component u_i will produce a new lower dimension dataset, on which a component C_j for $0 \leq j \leq p-1$ (p being the lower dimension number of components capturing the most variance) will be a weighted sum of the original dataset features:

$C_j = w_1(x_1) + w_2(x_2) + ... + w_n(x_n)$.

The steps are as follows:

Step 1: Center the dataset around zero means.

1. $X = \forall m \, (x_m - \bar{x})$

Step 2: Calculate the scatter matrix S_T of the dataset, as the covariance matrix multiplied by (m-1)

2. $S_T = \sum_{m=1}^{M} x_m x_m^T = \mathrm{XX^T}$

Step 3: Solve for the first component:

3. $\widehat{u_1} = \underset{u_1}{\operatorname{argmax}} u_1^T S_T u_1$ subject to $u_1^T u_1 = 1$ as a normalisation constraint.
4. This can be solved by Lagrange multiplier λ to include the constraint in the equation: $\psi = u_1^T S_T u_1 - \lambda(u_1^T u_1 - 1)$
   1. Then we minimise by differentiating with respect to u_1: and set to 0,

$$\frac{\delta\psi}{\delta u_1} = \mathrm{S}_T u_1 - \lambda u_1 = (\mathrm{S}_T - \lambda \mathrm{I}) u_1 = 0$$

Therefore, λ and u_1 are an Eigenvalue and its corresponding Eigenvector of S_T and the quantity to be maximised is: $u_1^T S_T u_1 = u_1^T \lambda u_1 = \lambda u_1^T u_1 = \lambda I = \lambda$. This will be the largest Eigenvalue and the corresponding Eigenvector as the first principal component.

Step 4: Solve for remaining p components by adding the orthogonality constraints by repeating steps 5:7 for each Eigenvector. For example, solving for u_2 require the following steps:

5. $\widehat{u_2} = \underset{u_2}{\operatorname{argmax}} u_2^T S_T u_2$ subject to $u_2^T u_2 = 1$ as a normalisation constraint and $u_2^T u_1 = 0$ as the orthogonality constraint with the previous Eigenvector.
6. Again use the Lagrange multiplier μ to include the second constraint in the equation: $\psi = u_2^T S_T u_2 - \lambda(u_2^T u_2 - 1) - \mu(u_2^T u_1)$
7. Differentiate with respect to u_2 and set = 0,

$$\frac{\delta\psi}{\delta u_2} = \mathrm{S}_T u_2 - \lambda u_2 - \mu u_1$$

8. Multiply on the left by $u_1^T = u_1^T \mathrm{S}_T u_2 - \lambda u_1^T u_2 - \mu u_1^T u_1 = 0 \rightarrow \mu = 0$
9. The first two terms are zero, as shown earlier and $u_1^T u_1 = 1$, leaving it $= (\mathrm{S}_T - \lambda \mathrm{I}) u_2 = 0$ as for u_1.

Step 5: Assemble the projection Matrix U columns, $\boldsymbol{u_1, u_2, \ldots, u_p}$

Step 6: Project the dataset y = Ux

PCA can be expressed in a more succinct matrix form. If E is the (orthonormal) matrix of column Eigenvectors of the covariance matrix C, and Λ the diagonal matrix of (non-negative) Eigenvalues of C, then CE = EΛ. The order of the columns of E needs to be such that the $\lambda_i \equiv \Lambda_{ii}$ is in descending order: $\lambda_i \leq \lambda_{i-1}\ \forall\ i = 1, \ldots, d-1$. Now for some unit vector $n_1 \in R^d$

consider the quantity: $n_1' E^T C E n_1 = n_1' \Lambda n_1$. The left-hand side is the variance of the projections of the (centred) data along the unit vector principal component En_1. The right-hand side is $\sum_i n_{1_i}^2 \lambda_i$.

The variance of the data projected along the direction of this principal component is just λ_i. This can be repeated until the required variance is captured by searching for the unit vector n_2 that is orthogonal to n_1 (i.e., for which $n_{21} = 0$) and which maximises the right-hand side, which is given by $n_{2i} = \delta_{i,2}$ (Burges, 2009).

Many PCA-based algorithms exist, including Kernel PCA, Probabilistic PCA, and oriented PCA. In summary, PCA projects the existing data on fewer linearly independent components capturing as much variance of the data as possible. When the remaining variance along all directions orthogonal to the p principal components chosen is zero or very negligible, then the dataset lies along a lower-dimensional manifold $\in \mathbb{R}^p$ embedded in R^m. (Lu, Plataniotis and Venetsanopoulos, 2014).

PCA Matrix example:

Find the PCA of a matrix $X \in \mathbb{R}^{4\times3} = \begin{bmatrix} 2 & 3 & 1 \\ 3 & 1 & 0 \\ 2 & 0 & 1 \\ 5 & 0 & 2 \end{bmatrix}$

Step 1: Data preprocessing: remove mean values from each feature

```
# First Step by Step calculations, but using implemented functions for mean,
covariance, and Eigendecomposition.
import numpy as np
X = np.array([[2,3, 1], [3, 1, 0], [2, 0, 1], [5, 0, 2]])
X
```

```
array([[2, 3, 1],

       [3, 1, 0],

       [2, 0, 1],

       [5, 0, 2]])
```

```
# Calculate the mean of each column
M = np.mean(X.T, axis=1)
M
```

```
array([3., 1., 1.])
```

```
#center columns by subtracting column means
C = X - M
C
```

```
array([[-1.,  2.,  0.],
       [ 0.,  0., -1.],
       [-1., -1.,  0.],
```

```
       [ 2., -1.,  1.]])
```

Step 2: Calculate the covariance matrix.

```
#calculate covariance matrix
sigma = np.cov(C.T)
sigma
```

```
array([[ 2.        , -1.        ,  0.66666667],
       [-1.        ,  2.        , -0.33333333],
       [ 0.66666667, -0.33333333,  0.66666667]])
```

Step 3: Calculate the eigenvectors and eigenvalues.

```
# eigen decomposition of covariance matrix
from numpy import linalg as LA
eignVal, eignVec = LA.eig(sigma)
eignVal, eignVec
```

```
(array([3.19940358, 1.0821583 , 0.38510479]),
 array([[-0.70083477,  0.57765664, -0.41850141],
        [ 0.65971771,  0.74802961, -0.07227894],
        [-0.27129904,  0.32674839,  0.90533548]]))
```

Step 4: Select Principal components. This small example did not reduce the dimensionality and selected all principal components. Sometimes you need to reduce the dimensionality to 2 or 3 PCs out of 10 or more features by capturing the highest variance by checking the "elbow" in the curve of a scree plot.

```
# Print principal components
PC1 = eignVec[:,0].T.dot(C.T) # u0.x
PC2 = eignVec[:,1].T.dot(C.T) # u1.x
PC3 = eignVec[:,2].T.dot(C.T) # u2.x
print("PC1: ", PC1)
print("PC2: ", PC2)
print("PC3: ", PC3)
```

```
PC1:  [ 2.02027019  0.27129904  0.04111706 -2.33268629]

PC2:  [ 0.91840258 -0.32674839 -1.32568625  0.73403206]

PC3:  [ 0.27394352 -0.90533548  0.49078035  0.1406116 ]
```

Step 5: Derive the new projected dataset: Projected_data = normalised_data x selected_principal_components.

```
# project data
P = eignVec.T.dot(C.T)
P.T
```

```
array([[ 2.02027019,  0.91840258,  0.27394352],
       [ 0.27129904, -0.32674839, -0.90533548],
       [ 0.04111706, -1.32568625,  0.49078035],
```

```
        [-2.33268629,  0.73403206,  0.1406116 ]])
```

```
# Using Sklearn PCA implementation
#Second numpy Calculations
from sklearn.decomposition import PCA

#create PCA instance
pca = PCA(3)
#fit on Data
pca.fit(X)
#access values and vectors
print("PCA Components = ", pca.components_)
print("PCA Explained variance ", pca.explained_variance_)
print("PCA Explained variance ratio ",
pca.explained_variance_ratio_)
B = pca.transform(X)
print("Transformed Data  ", B)
```

```
PCA Components =  [[ 0.70083477 -0.65971771  0.27129904]
 [-0.57765664 -0.74802961 -0.32674839]
 [ 0.41850141  0.07227894 -0.90533548]]
PCA Explained variance  [3.19940358 1.0821583  0.38510479]
PCA Explained variance ratio  [0.68558648 0.23189106 0.08252245]
Transformed Data   [[-2.02027019 -0.91840258 -0.27394352]
 [-0.27129904  0.32674839  0.90533548]
 [-0.04111706  1.32568625 -0.49078035]
 [ 2.33268629 -0.73403206 -0.1406116 ]]
```

2.2.4 Singular Value Decomposition (SVD)

SVD decomposes a matrix X into three constituent matrices to remove the redundancy in the original features by choosing the highest singular values and their corresponding features resulting in dimensionality reduction. The Gauss-Jordan Elimination in chapter one showed that the diagonalisation of a matrix reduces it to an easy-to-analyse matrix. The Eigendecomposition takes the form Au_i = $\lambda_i u_i$ where u_i is the eigenvector corresponding to the λ_i eigenvalue. This can be expressed in diagonalised form as: AU =ΛU or A =$U\Lambda U^{-1}$ such that Λ is a diagonal matrix containing the eigenvalues on the diagonal elements, and U contains the eigenvectors as its columns (Deisenroth, Faisal and Ong, 2019).

$$A = \begin{bmatrix} u_{0,0} & \cdots & u_{o,n-1} \\ \vdots & \ddots & \vdots \\ u_{n-1,0} & \cdots & u_{n-1,n-1} \end{bmatrix} \begin{bmatrix} \lambda_0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_{n-1} \end{bmatrix} \begin{bmatrix} u_{0,0} & \cdots & u_{o,n-1} \\ \vdots & \ddots & \vdots \\ u_{n-1,0} & \cdots & u_{n-1,n-1} \end{bmatrix}^{-1}$$

Where $u_i = [u_{o,i}, u_{1,i}, \ldots, u_{n-1,i}]$

The inverse Eigenvectors matrix exists only if the Eigenvectors are linearly independent. This is possible only if A is a square matrix. To develop a similar decomposition for non-square matrices, SVD starts by converting the input matrix to a special square matrix by multiplying it by its transpose. A^TA matrix has the following properties:

- Symmetrical, therefore, we can choose its eigenvectors to be orthogonal with unit length (orthonormal) to each other.
- Square, i.e. invertible
- At least positive semi-definite (Eigenvalues are zero or positive),
- Both matrices (A and A^TA) have the same positive Eigenvalues, and
- Both have the same rank r as A.

The equation to decompose a matrix X is given by: $X_{mxn} = U_{mxn}\mathrm{S}_{nxn}V^T_{nxn}$

Where

- **S** is an n×n diagonal matrix, and the diagonal values in the **S** matrix are known **as the singular values** of the original matrix **X**.
    - The singular values are the square root of the Eigenvalues. The singular values are arranged in descending order along the main diagonal in **S**.
    - It is a diagonal matrix that can be reduced to only $\boldsymbol{r}$ important values, the rest of the matrix being zero. The choice of the rank is based on eliminating linear dependant rows, which is the default matrix rank definition and can be any smaller value than n to achieve lossy compression (leaving out some non-zero but small Eigenvalues).
- $\mathbf{V}$ is an orthonormal column matrix; the columns of $\mathbf{V}$ are called **the right singular vectors** of **X**
    - Note that we always use V in its transposed form, so the rows of V^T are orthonormal.
    - It is a matrix that holds important information about the columns of **X**, and the most important information about $\mathbf{V^T}$ is stored in the first row.
    - V is the corresponding unit Eigenvectors to the Eigenvalues squared and placed in S.
- $\mathbf{U}$ is an orthonormal column matrix; the columns of the $\mathbf{U}$ matrix are called **the left singular vectors** of **X**;
    - Each of its columns is a unit vector, and the dot product of any two columns is 0 (orthogonal).
    - It is a matrix that holds important information about the rows of **X**, and the most important information about $\boldsymbol{U}$ is stored in the first column.

- It is calculated by using the property US(:,i) = XV(:,i)

For compression, we define the rank for the middle projection:

$$X_{mxn} = U_{mxr}S_{rxr}V^T_{rxn}$$

Where r is rank $r \leq$ m and $r \leq$ n, the rank of a matrix is the largest number of columns we can choose for which no selected column is a linear combination of another selected column. This is to say that such columns are linearly independent. This can also be expressed in vectors form as linear combinations of orthonormal basis directions weighted by the singular value σ in descending order: $X = USV^T = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_r u_r v_r^T$. This is illustrated in Figure 2.

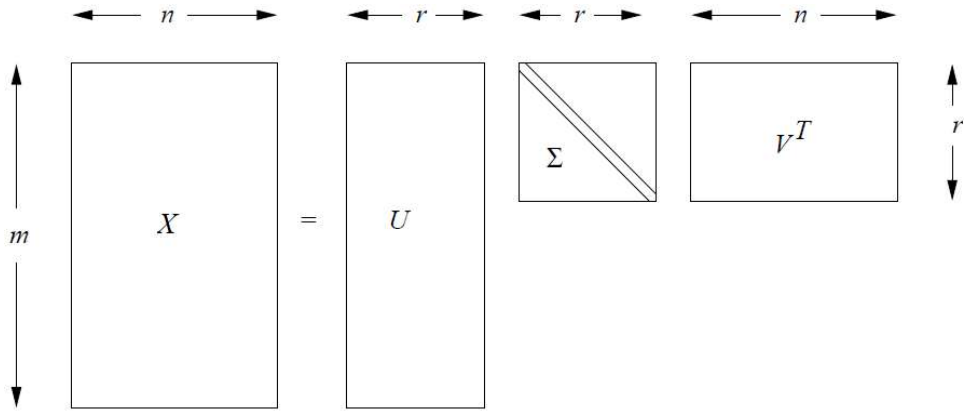


Figure 2: SVD matrix decomposition illustration

Steps to calculate SVD:

Given a matrix X:

Step 1: Compute A=X^TX

Step 2: Do Eigendecomposition of A

Step 3: Calculate the singular values, which are the square root of the Eigenvalues

1. Sort the singular values in decreasing order; and arrange in S or Σ matrix along the diagonal.

Step 4: Calculate the right singular vectors (columns of V)

1. the corresponding unit Eigenvectors are the right singular vectors of X, stored in V

Step 5: Find the left singular vectors U by using the property US(:,i) = MV(:,i)

Example 1:

Find Singular Value Decomposition (SVD) of a Matrix M ∈ $\mathbb{R}^{m\times r}$ = $\begin{bmatrix}-1 & 0\\ 0 & 3\end{bmatrix}$

SVD(M) = U Σ V^T

Step 1: Calculate A = M^TM = $\begin{bmatrix}-1 & 0\\ 0 & 3\end{bmatrix}\begin{bmatrix}-1 & 0\\ 0 & 3\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0 & 9\end{bmatrix}$

Step 2: Find Eigenvalues and Eigenvectors of A = M^TM

$|A - \lambda I| = 0$

$\begin{bmatrix}1-\lambda & 0\\ 0 & 9-\lambda\end{bmatrix} = 0$

Then arranging by highest Eigenvalue, we have $\lambda_0 = 9,\ \lambda_1 = 1$

For $\lambda_0 = 9$, the corresponding Eigenvector is:

$\begin{bmatrix}1-9 & 0\\ 0 & 9-9\end{bmatrix}\begin{bmatrix}x_0\\ x_1\end{bmatrix} = 0$

$\begin{bmatrix}-8 & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}x_0\\ x_1\end{bmatrix} = 0$

Now, reduce this matrix as follows: (check the Gaussian Elimination in chapter one)

R1←R1÷-8

$\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}x_0\\ x_1\end{bmatrix} = 0$

Then $x_0 = 0\ and\ x_1 = 1$

Then $\boldsymbol{v_0} = \begin{bmatrix}\mathbf{0}\\ \mathbf{1}\end{bmatrix}$ is the Eigenvector corresponding to the Eigenvalue $\lambda_0 = 9$

For $\lambda_1 = 1$, the corresponding Eigenvector is:

$\begin{bmatrix}1-1 & 0\\ 0 & 9-1\end{bmatrix}\begin{bmatrix}x_0\\ x_1\end{bmatrix} = 0$

$\begin{bmatrix}0 & 0\\ 0 & 8\end{bmatrix}\begin{bmatrix}x_0\\ x_1\end{bmatrix} = 0$

Now, reduce this matrix by swapping the rows

$\begin{bmatrix}0 & 8\\ 0 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_0\end{bmatrix}$ = 0

Then, reduce this matrix as follows:

R1←R1÷-8

$\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_0\end{bmatrix}$ = 0

Then $x_0 = 1\ and\ x_1 = 0$

Then $\boldsymbol{v_1} = \begin{bmatrix}\mathbf{1}\\ \mathbf{0}\end{bmatrix}$ is the Eigenvector corresponding to the Eigenvalue $\lambda_1 = 1$

Step 3: Calculate the singular values, which are the square root of the eigenvalues

$\sigma_0 = \sqrt{\lambda_0} = \sqrt{9} = 3$

$\sigma_1 = \sqrt{\lambda_1} = \sqrt{1} = 1$

This forms your singular values diagonal matrix $\Sigma \in \mathbb{R}^{r\times r} = \begin{bmatrix}\sigma_0 & 0\\ 0 & \sigma_1\end{bmatrix} = \begin{bmatrix}3 & 0\\ 0 & 1\end{bmatrix}$

Step 4: Calculate the right singular vectors (columns of V)

V is the corresponding unit eigenvectors, now called the right singular vectors of M

$V \in \mathbb{R}^{r\times n} = [v_0\ \ v_1] = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$

PS $V^TV = I$ (V has orthonormal columns: orthogonal and unit bases – of length 1).

Step 5: Find the left singular vectors U by using the property US(:,i) = MV(:,i)

Which is $u_0 = \frac{1}{\sigma_0}M^T.v_0 = \frac{1}{3}\begin{bmatrix}-1 & 0\\ 0 & 3\end{bmatrix}.\begin{bmatrix}0\\ 1\end{bmatrix} = \begin{bmatrix}\frac{-1}{3} & 0\\ 0 & 1\end{bmatrix}.\begin{bmatrix}0\\ 1\end{bmatrix} = \begin{bmatrix}0\\ 1\end{bmatrix}$

And $u_1 = \frac{1}{\sigma_1}M^T.v_1 = \frac{1}{1}\begin{bmatrix}-1 & 0\\ 0 & 3\end{bmatrix}.\begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}-1 & 0\\ 0 & 3\end{bmatrix}.\begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}-1\\ 0\end{bmatrix}$

$U \in \mathbb{R}^{m\times r} = [u_0 \quad u_1] = \begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix}$

PS $U^TU = I$ (U has orthonormal columns: orthogonal and unit bases – of length 1).

The complete solution is: $SVD(M) = U\ \Sigma\ V^T = \begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix}\begin{bmatrix}3 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$

<u>Example 2:</u>

Given $X = \begin{bmatrix} -2 & 2 \\ -4 & -4 \\ 4 & 4 \\ 2 & -2 \\ 0 & 0 \end{bmatrix}$

To compute SVD:

Step 1: Compute M=$X^T X$=$\begin{bmatrix} 40 & 24 \\ 24 & 40 \end{bmatrix}$

Step 2: do Eigendecomposition of M:

Eigenvalues: $\lambda_1 = 64$ and $\lambda_2 = 16$

Eigenvectors: $u_1^T \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}, u_2^T \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$

Step 3: Find Σ. From the square roots of the eigenvalues on the diagonal

$s_1=\sqrt{64}=8$, $s_2=\sqrt{16}=4$

$\Sigma = \begin{bmatrix} 8 & 0 \\ 0 & 4 \end{bmatrix}$

Step 4: Find V. From the eigenvectors

$V=\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

Step 5: Find U. For example:

$$u_1 = \frac{1}{s_1} \times M^T . v_1 = \frac{1}{8}\begin{bmatrix} 40 & 24 \\ 24 & 40 \end{bmatrix}\begin{bmatrix} -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix},$$

$$u_2 = \frac{1}{s_2} \times M^T . v_2 = \frac{1}{4}\begin{bmatrix} 40 & 24 \\ 24 & 40 \end{bmatrix}\begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$U = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Python example:

```
from numpy import array
from scipy.linalg import svd, diagsvd

X = array([[1, 2], [3, 4], [5, 6]])
print (X)
U, S, VT = svd(X)
# to reconstruct  the matrix:
Sigma = diagsvd(S, X.shape[0], X.shape[1])
X_reconstructed = np.dot(U, np.dot(Sigma, VT))
print(X_reconstructed)
np.allclose(X, X_reconstructed)
```

```
X =  [[1 2]

 [3 4]

 [5 6]]


U =  [[-0.2298477   0.88346102  0.40824829]

 [-0.52474482  0.24078249 -0.81649658]

 [-0.81964194 -0.40189603  0.40824829]]


S =  [9.52551809 0.51430058]


VT =  [[-0.61962948 -0.78489445]

 [-0.78489445  0.61962948]]


X Reconstructed =  [[1. 2.]

 [3. 4.]

 [5. 6.]] Allclose: True
```

2.2.4.1 Data Science with SVD

This example is credited to (Leskovec, Rajaraman and Ullman, 2014). Given Ratings of five movies by seven users. The movies belong to a category or concept, and this information is hidden in the observed data. We know that the movies belong to 2 categories: science fiction and romance, so we choose *r* equals 2 to reveal these hidden (latent) features. We can do SVD to identify the users' interest in the concept in general and evaluate the weights of ratings by the concept, such that:

$\boldsymbol{M^{n\times d}=U^{n\times r}\Sigma^{r\times r}\,(V^{r\times d})^{T}}$

$\boldsymbol{M}$: $n$umber of users (n) x number of movies (d)

$\boldsymbol{U}$: n users x r concepts: connects people to concepts.

$\boldsymbol{\Sigma}$: the strength of each concept

$\boldsymbol{V}$: d movies x r concepts: relates movies to concepts

	Matrix	Alien	Star Wars	Casablanca	Titanic
Joe	1	1	1	0	0
Jim	3	3	3	0	0
John	4	4	4	0	0
Jack	5	5	5	0	0
Jill	0	0	0	4	4
Jenny	0	0	0	5	5
Jane	0	0	0	2	2

Doing SVD on this matrix:

$$\begin{bmatrix}1 & 1 & 1 & 0 & 0\\ 3 & 3 & 3 & 0 & 0\\ 4 & 4 & 4 & 0 & 0\\ 5 & 5 & 5 & 0 & 0\\ 0 & 0 & 0 & 4 & 4\\ 0 & 0 & 0 & 5 & 5\\ 0 & 0 & 0 & 2 & 2\end{bmatrix} = \begin{bmatrix}0.14 & 0\\ 0.42 & 0\\ 0.56 & 0\\ 0.70 & 0\\ 0 & 0.60\\ 0 & 0.75\\ 0 & 0.30\end{bmatrix} \begin{bmatrix}12.4 & 0\\ 0 & 9.5\end{bmatrix} \begin{bmatrix}0.58 & 0.58 & 0.58 & 0 & 0\\ 0 & 0 & 0 & 0.71 & 0.71\end{bmatrix}$$

$$X = U \boldsymbol{\Sigma} V^T$$

Note: Generally, the concepts will not be as clearly delineated as in this hand-crafted example. There will be fewer zeros in $\boldsymbol{U}$and $\boldsymbol{V}$. Check the ch2.ipynb to see actual values generated from the sklearn Python package implementation of "TruncatedSVD".

We can read various information from the SVD decomposition:

- **V^T:** the first row of V^T says that the first three movies belong to science fiction, while the second row of V^T tells us that the last two movies belong to the romance genre.
- **U:** We can tell that Joe, Jim, John, and Jack are interested in science fiction only, while Jill, Jenny, and Jane are interested in romance only and do not rate science fiction movies at all.
- $\boldsymbol{S}_1$: the strength of the first concept: sci-fic;
- $\boldsymbol{X}V_1$: each user's rating is an average of all five movies, weighted by the first concept;
- $\|\boldsymbol{X}V_1\|$: a score by the overall 7 users, weighted by the first concept.
- $S_i = \|\boldsymbol{X}V_i\|$, where $\boldsymbol{S_i}$ is the i^{th} value along the main diagonal line of $\boldsymbol{S}$, and $\boldsymbol{V_i}$ is the i^{th} column of $\boldsymbol{V}$.

Online SVD Calculators:

https://atozmath.com/MatrixEv.aspx?q=svd

https://keisan.casio.com/exec/system/15076953160460

2.2.4.2 Compression with SVD

Other applications of SVD ($X_{nxd} = U_{nxr} S_{rxr} V^T_{rxd}$) is to choose the highest rank-r approximation to a given matrix X that is given by $X_{nxd} = AW^T$, such that A =$U_{nxr}$and W = $S_{rxr} V^T_{rxd}$. The choice of small rank can be useful for compression. This happens by setting the s=n-r smallest singular values to 0; then we can also eliminate the corresponding s columns of U and V. Choosing the lowest singular values to drop minimises the root-mean-square error between the original matrix X and its reconstructed approximation. The choice of r can be made by studying the decay of the singular values. Python notebook ch2.ipynb shows an image compression SVD example and how the choice of rank affects the quality of the image.

2.2.5 Independent Component Analysis (ICA)

Independent Component Analysis (ICA) is a generative model that assumes a dataset to be a linear mixture of some latent factors such that x = As, where A∈ $\mathbb{R}^{p\times p}$ is the mixing matrix, and s is the source or the independent components. ICA uses information theory to capture mutually independent factors in a dataset, unlike PCA, which captures uncorrelated factors. ICA estimates p unordered sources that should not be more than the observed mixtures x (Lu, Plataniotis and Venetsanopoulos, 2014).

ICA attempts to find the unmixing matrix by employing non-Gaussianity from the central tendency theorem to retrieve the independent components from the mixture: $s = A^{-1}x$. One possible computational set of steps to estimate ICs is as follows:

Step1: Normalise and Whitening:

1. Subtract the mean: $x_{norm}=x-\bar{x}$
2. Remove the correlations by Eigendecomposition to end up with zero covariance and 1 variance: $x_w=(ED^{-\frac{1}{2}}E^T)x$, Where, D is a diagonal matrix of eigenvalues, every diagonal element is an eigenvalue of the covariance matrix, and E is an orthogonal matrix of eigenvectors.

Step 2: Maximise Non-Gaussainty by Negative Entropy

3. $A=(VD^{-\frac{1}{2}}E^T)$, where V is an unknown rotation matrix, that we need to solve for it and A using the eigenvalues and eigenvectors. Solving for two unknown iteratively by Lagrange multiplier for the constraints that the dot product of transpose of W and itself is approximately equal to 1: $A^TA\approx1$, and the Newton iterations by randomly initialising W with any values, and iterate using an objective function objFunc that aims to non-Gaussian maximisation such as tanh and its derivative dObjFunc as follows:
4. $w\_new=\frac{1}{n}\sum_{i=1}^{n}X\times objFunc(A^T.X)-\frac{1}{n}\sum_{i=1}^{n}dObjFunc(A^T.X)\times A$

Step 3: Define the convergence scheme:

1. A is initialised to a random variable
2. $A^TA\approx1$ is the basis of orthogonality and indicates the convergence
3. Once the resulting matrix A is calculated, the dot product of it and the whitened x_w signal gives the sources s.

There are other algorithms to perform ICA such as infomax, FastICA, and JADE.

Real-World Example:

ICA is important for Blind Source Separation (BSS) problems. A classical BSS problem is the cocktail party problem (CPP), defined as having two microphones in a room in different locations, while two people are talking simultaneously. The mixed-time-signals received at both microphones are $x_1(t) = a_{11}s_1(t) + a_{12}s_2(t) and\ x_2(t) = a_{21}s_1(t) + a_{22}s_2(t)$, such that a_{ij} are the weight parameters dependent on the distances between microphone i and speaker j collected in matrix A, in this case, it is a 2x2 matrix but could be expanded to n microphone and m sources problem. The aim is to estimate s_1 and s_2 from both equations with only $x_1(t)$ and $x_2(t)$ being known. Check the Python code ICA_FOBI.ipynb for an audio BSS example, followed by an image mixture separation example.

2.2.6 Linear Discriminant Analysis (LDA)

LDA is a supervised dimensionality reduction algorithm that works on labelled data to discriminate between classes, unlike unsupervised PCA, which diagonalises the covariance matrix to make data points independent of each other. The most straightforward Binary (2 classes) Linear Discriminant is based on the mean of both classes $\mu_{c_i}^k$ and their variance $s_{c_i}^k$, creating a metric $J_k = \frac{\left(\mu_{c_1}^k - \mu_{c_2}^k\right)^2}{\left(s_{c_1}^k\right)^2 + \left(s_{c_2}^k\right)^2}$, where k spans the features/columns assuming only two features, i spans the classes (assuming two classes), and the discriminate becomes identifying the value of k at which the highest value of J_k is found. Then, taking the classes' means as the threshold draws the line that separates both classes $c = \frac{\mu_{c_1}^k + \mu_{c_2}^k}{2}$. To generalise for multiple features to draw a discriminant line equation, y=wx, we need to solve for w, through the following steps:

1. Calculate a middle point of all n features from the means of both classes as a vector, $M = \left(\frac{\mu_{c_1}^{f_1} + \mu_{c_2}^{f_1}}{2}, \ldots \frac{\mu_{c_1}^{fn} + \mu_{c_2}^{fn}}{2}\right)$,
2. Calculate the covariance matrices of both classes, S_1, S_2 and compute their average $S_{avg} = \frac{1}{2}[S_1 + S_2]$
3. Solve for weights vector from the inverse of the average, $w = S_{avg}^{-1}\left(\mu_{c_1} + \mu_{c_2}\right)$
4. Identify the threshold c using the middle point, c=wm
5. For every new test sample x, solve for y=wx, if y > c, then x belongs to c_1, and when y < c, the sample belongs to c_s.

Another method to divide the data into separate classes is to maintain two scatter matrices for the input space: the within-class scatter S_w to maximise and the between-class scatter S_b to minimise for a dataset X with M instances and C classes. The scatter matrices are covariance matrices multiplied by the number of entities (Lu, Plataniotis and Venetsanopoulos, 2014).

The steps to perform LDA using scatter matrices are as follows:

Step 1: Calculate the scatter matrices for the input space:

1. $S_w = \sum_{i=1}^{M}(x_i - \bar{x}_{c_i})(x_i - \bar{x}_{c_i})^T$ and $S_b = \sum_{i=1}^{C} M_c(x_i - \bar{x})(x_i - \bar{x})^T$, where $\bar{x}_{c_i} = \frac{1}{M_c}\sum_{i\ in\ M_c}^{M_c} x_i$, is the class mean for entities belonging to the class Mc, and $\bar{x}$ is the feature column mean across all classes. The sum of these two scatter matrices is the total scatter S_t for the dataset.

Step 2: Estimate the projection matrix into the output space:

2. The input space scatter-matrices are projected into the output space using projection matrix U as follows:

   $S_{wY} = U^T S_w U$, and $S_{bY} = U^T S_b U$ and the Total scatter in the output space: $S_{tY} = U^T S_t U$

3. LDA aims to maximise the between-class scatter in the output space S_{bY}, while minimising the within-class scatter in the output space S_{wY} by solving for U. This can be done by using the trace of the scatter matrices and the inverse if it is singular:

$$\widehat{U} = \underset{U}{\operatorname{argmax}}\, tr(S_{wY}^{-1} S_{bY}) = \underset{U}{\operatorname{argmax}}\, tr((U^T S_w U)^{-1} U^T S_b U)$$

- It can also be done using any substitutions of S_{tY}, or by using the determinants:

$$\widehat{U} = \underset{U}{\operatorname{argmax}} \frac{|S_{bY}|}{|S_{wY}|} = \underset{U}{\operatorname{argmax}} \frac{|U^T S_b U|}{|U^T S_w U|}$$

$$\widehat{U} = \underset{U}{\operatorname{argmax}}\, U^T S_b U, \text{ subject to } U^T S_w U = 1.$$

4. This can be solved by Lagrange multiplier: $\psi = U^T S_b U - \lambda(U^T S_w U - 1)$, then differentiate with respect to u, and set = 0,

$$\frac{\delta\psi}{\delta u} = S_b U - \lambda S_w U = 0 \rightarrow S_b U = \lambda S_w U$$

   which is the generalised eigenvalue problem.

Step 3: Get the highest C-1 eigenvectors to form U:

5. The eigenvectors corresponding to the largest C-1 eigenvalues λ form the columns of U.

Step 4: Project the test data into the new LDA output space:

6. Then data are projected like in PCA as $y = U^T x$.

The above approaches build discriminative classifiers by learning to discriminate the classes from a training dataset, not how the data points are generated. A generative classifier uses probabilistic methods to learn how points are generated in each class, applying Bayes Theorem. LDA is implemented in Scikit learn Python packages using its generative model. The LDA generative model assumes the class-conditional densities, $f_k(x)$ for class k, are Gaussian with a common covariance matrix so that $f_k(x) = \frac{1}{(2\pi)^{M/2}|\Sigma|^{1/2}} e^{\frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k)}$, where the Gaussian is described using its mean and standard deviation, or covariance matrix Σ. The function includes constant e, M is the number of features, and μ_k is means for class k. Given N_k as the number of entities in class k, such that the LDA classifier assumes that $\hat{\pi}_k = \frac{N_k}{N}$ is the maximum likelihood estimates (MLE) of $f_k(.)$, across all classes, i.e. π_k is the probability of class Y being k, P(Y = k). The $f_k(x)$ predictors (for k=1,…,K) all share the same covariance matrix $\hat{\Sigma} = \frac{\sum_{k=1}^{K}\sum_{y_i=k}(x-\hat{\mu}_k)(x-\hat{\mu}_k)^T}{N-k}$ but may have different means $\hat{\mu}_k = \sum_{yi=k}\frac{x_i}{N_k}$. The Bayes Theorem: $\hat{Y} = \underset{Y}{\operatorname{argmax}}\,\hat{P}(Y|X) = \underset{Y}{\operatorname{argmax}}\,\hat{P}(X|Y)\hat{P}(Y)$, predicts the class $\hat{Y}$ of test sample X. The generative LDA classifier satisfies: $\hat{Y}(x) = \underset{k}{\operatorname{argmax}}\frac{\hat{f}_k(x)\hat{\pi}_k}{\sum_{j=1}^{k}\hat{f}_j(x)\hat{\pi}_j}$.

LDA can separate multiple classes by training multiple classifiers to separate each pair of classes and use voting between the classifiers to identify the class of a test sample, which can be considered a basic multilinear model. Instead of using a linear equation, a quadratic equation is used in QDA. The Bayesian Probabilistic models of LDA and QDA are implemented in Scitkit-learn library, assuming Gaussian densities for $\hat{P}(X|Y)$, while the other generative model, such as Naive Bayes, assumes $P(X|Y) = \prod_j P(X_j|Y)$, such that the features are independent conditional on Y. For more about Bayesian generative models, read books such as (Sivia and Skilling, 2006), (Downey, 2013). Check the Python code (**xxxx**) for an LDA example compared to PCA.

2.2.7 Canonical Correlation Analysis (CCA)

The CCA and the PLS (in the following section) algorithms consider mapping from two paired datasets, x ∈ $\mathbb{R}^{I\times M}$ and y∈ $\mathbb{R}^{J\times M}$. Learning the symmetric relationship between them is performed by learning the Linear projections for both by maximising the correlations.

Symmetric relationships mean both datasets are different views or representations of the same entities. A classic example is audio and video datasets from the same individuals (Lu, Plataniotis and Venetsanopoulos, 2014).

The steps to perform CCA are as follows:

Step 1: Define the covariance matrix of each dataset and the cross-covariance matrices:

1. $\Sigma_{xx} = \frac{1}{M-1}\sum_{i=1}^{M}(x_i - \bar{x})(x_i - \bar{x})^T$, $\Sigma_{yy} = \frac{1}{M-1}\sum_{i=1}^{M}(y_i - \bar{y})(y_i - \bar{y})^T$
   $\Sigma_{xy} = \frac{1}{M-1}\sum_{i=1}^{M}(x_i - \bar{x})(y_i - \bar{y})^T$, $\Sigma_{yx} = \frac{1}{M-1}\sum_{i=1}^{M}(y_i - \bar{y})(x_i - \bar{x})^T$, note that $\Sigma_{xy} = \Sigma_{yx}^T$

Step 2: Find The correlations between the projections:

2. The p[th] pair of projections ($u_{x_p}^T$, $u_{y_p}^T$), $r_{m_p} = u_{x_p}^T x_m$, and $s_{m_p} = u_{y_p}^T y_m$ such that coordinate vectors $w_p = u_{x_p}^T(x_i - \bar{x}) = r_{m_p}$, and $z_p = u_{y_p}^T(y_i - \bar{y}) = s_{m_p}$ has the correlation $\rho_p = \frac{u_{x_p}^T \Sigma_{xy} u_{y_p}}{\sqrt{(u_{x_p}^T \Sigma_{xx} u_{x_p})(u_{y_p}^T \Sigma_{yy} u_{y_p})}}$

Step 3: Find the first projection pair $\{u_{x1}, u_{y1}\}$ such that ρ_1 is maximised similar to the algorithm presented in the PCA section,

3. $(u_{x1}, u_{y1}) = \underset{(u_{x1}, u_{y1})}{\operatorname{argmax}} u_{x_1}^T \Sigma_{xy} u_{y_1}$, subject to $u_{x_1}^T \Sigma_{xx} u_{x_1} = 1$, and $u_{y_1}^T \Sigma_{yy} u_{y_1} = 1$.
4. Using Lagrange Multipliers: $\psi_1 = u_{x_1}^T \Sigma_{xy} u_{y_1} - \frac{1}{2}\lambda\left(u_{x_1}^T \Sigma_{xx} u_{x_1} - 1\right) - \frac{1}{2}\mu\,(u_{y_1}^T \Sigma_{yy} u_{y_1} - 1)$
5. Differentiate with respect to u_{x1} and u_{y1}, and set to 0:
   $\frac{\delta\psi}{\delta u_{x1}} = \Sigma_{xy} u_{y_1} - \lambda \Sigma_{xx} u_{x_1} = 0$, then multiply $u_{x_1}^T$ on the left
   $\frac{\delta\psi}{\delta u_{y1}} = \Sigma_{yx} u_{x_1} - \mu \Sigma_{yy} u_{y_1} = 0$, then multiply $u_{y_1}^T$ on the left
6. We have $u_{x_1}^T \Sigma_{xy} u_{y_1} - \lambda u_{x_1}^T \Sigma_{xx} u_{x_1} = 0$, and $u_{y_1}^T \Sigma_{yx} u_{x_1} - \mu u_{y_1}^T \Sigma_{yy} u_{y_1}$
7. Since we have $u_{x_1}^T \Sigma_{xx} u_{x_1} = 1$, and $u_{y_1}^T \Sigma_{yy} u_{y_1} = 1$, then we are $\lambda = \mu = u_{x_1}^T \Sigma_{xy} u_{y_1}$ to maximise.
8. If Σ_{xx} is non-singular, $u_{x_1} = \frac{\Sigma_{xx}^{-1}\Sigma_{xy}}{\lambda} u_{y_1}$, then we have: $\Sigma_{yx}\Sigma_{xx}^{-1}\Sigma_{xy} u_{y_1} = \lambda^2 \Sigma_{yy} u_{y_1}$, which is the generalised eigenvalue problem, such that $\hat{u}_{y_1}$ is the Eigenvector corresponding to the largest Eigenvalue
9. If Σ_{yy} is non-singular, then we have: $\Sigma_{yy}^{-1}\Sigma_{yx}\Sigma_{xx}^{-1}\Sigma_{xy} u_{y_1} = \lambda^2 u_{y_1}$, and $\Sigma_{xx}^{-1}\Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{yx} u_{x_1} = \lambda^2 u_{x_1}$

Step 4: Find the following projections

10. Each projection p should maximise the correlation ρ_p by adding the zero correlation constraints such as for p=2, w_2 and z_2 are uncorrelated with w_1 and z_1, and for later steps p, w_p and z_p are uncorrelated with w_q and z_q, for 1<= q <= p-1

Step 5: Assemble the Projection Matrices

11. Projections matrices $U_x \in \mathbb{R}^{I \times p}$ and $U_y \in \mathbb{R}^{J \times p}$columns are the calculated projection pairs, respectively.
12. Make the projections to the dataset as $x_{projected} = U_x^T x$, and $y_{projected} = U_y^T y$.

Check the Python code in CCA_PLS.ipynb for a CCA example compared to PLS.

2.2.8 Partial Least Squares Analysis (PLS)

It is like CCA in mapping the symmetric relationship between two paired datasets, but by maximising the covariance, not the correlation, and is useful when Σ_{xx} and Σ_{yy} are not singular. In the scikit-learn Python package, this is implemented in the cross decomposition module as PLSCanonical. Maximising the covariance between datasets is suitable when modelling X as one dataset and Y as another dataset, such that the target is not a single value but a vector of several targets as multilabel. PLS is also an acronym for "Projection to Latent Structures" since it uses latent variables to identify the association between blocks of observed variables.

The p^{th} pair of projections (u_{x_p} , u_{y_p}) are the weight vectors that are estimated such that coordinate vectors $w_p = u_{x_p}^T(x_i - \bar{x}) = r_{m_p}$, and $z_p = u_{y_p}^T(y_i - \bar{y}) = s_{m_p}$ maximise the sample covariance between $\{w_p, z_p\}$, which are the score vectors, forming the **score Matrices** W and Z. This is calculated by: $\left(\hat{u}_{x_p}, \hat{u}_{y_p}\right) = \underset{u_{x_p}, u_{y_p}}{\operatorname{argmax}} \frac{u_{x_p}^T \Sigma_{xy} u_{y_p}}{\sqrt{(u_{x_p}^T u_{x_p})(u_{y_p}^T u_{y_p})}}$. The non-linear iterative partial least squares (NIPALS) algorithm iteratively finds the dominant Eigenvector similar to the power method. By deflating the X, and Y matrices, the algorithm finds the subsequent dominant Eigenvector (Lu, Plataniotis and Venetsanopoulos, 2014).

The steps to perform PLS are as follows:

Step 1: Repeat Until Convergence:

1. Initialise Y-score vector z_p^T with the first row of Y
2. Calculate the X-weight vectors: $u_{x_p} = \frac{X_{z_p}}{\left\|X_{z_p}\right\|}$,
3. Calculate the X-score vector: $w_p = X^T u_{x_p}$
4. Calculate the Y-weight vector: $u_{y_p} = \frac{Y_{w_p}}{\left\|Y_{w_p}\right\|}$

5. Update Y-score vector $z_p = Y^T u_{y_p}$
6. Repeat steps 1:5 until $\frac{\|\text{old } w_p - \text{new } w_p\|}{\|\text{new } w_p\|}$< stopping criteria η, u_{x_p} and u_{y_p} are the dominant eigenvectors of this round.
7. Calculate the X loading vector $v_{x_p} = \frac{X_{w_p}}{w_p^T w_p}$, coefficients of regressing X on w_p forming matrix V_x
8. Calculate the Y-loading vector $v_{y_p} = \frac{Y_{z_p}}{z_p^T z_p}$, coefficients of regressing Y on z_p forming matrix V_y
9. Rank-one deflate both X and Y: $X \leftarrow X - v_{x_p} w_p^T$ and $Y \leftarrow Y - v_{y_p} z_p^T$
10. Repeat steps 1:9 until $\|X\|$< stopping criteria η.

Step 2: Assemble the weight Matrices

11. Weight matrices $U_x \in \mathbb{R}^{I \times p}$ and $U_y \in \mathbb{R}^{J \times p}$ columns are the calculated projections pairs, respectively.
12. Make the projections to the dataset as $x_{projected} = U_x^T x$, and $y_{projected} = U_y^T y$.

PLS can be used for regression when the relationship between X and Y is asymmetric, and considering Y the response (outcome or dependent) on X (the predictor or the independent dataset), such that the score vectors $\{w_p\}$ are good predictors of Y. The residual matrices (**the error terms**) are E_x, E_y, such that $X = V_x W^T + E_x$, and $Y = V_y Z^T + E_y$, assembled in matrix E can be used to linearly approximate the relationship between W and Z, such that Z = WD + E, and D are the regression coefficients calculated below. PLS for regression performs the following steps:

Step 1: Repeat for p latent factors:

1. Calculate the X-weight vectors: $\hat{u}_{x_p} = \frac{X_y}{\|X_y\|}$,
2. Calculate the X-score vector: $w_p = X^T \hat{u}_{x_p}$
3. Calculate the regression coefficient $d_p = \frac{w_p^T y}{(w_p^T w_p)}$
4. Calculate the X loading vector $v_{x_p} = \frac{X_{w_p}}{w_p^T w_p}$, coefficients of regressing X on w_p forming matrix V_x
5. Rank-one deflate both X and Y: $X \leftarrow X - v_{x_p} w_p^T$ and $Y \leftarrow Y - d_p w_p$

Step 2: Assemble the weight Matrix, loadings, and regression coefficients

6. Weight matrices $U_x \in \mathbb{R}^{I \times p}$ columns are the calculated weights $\hat{u}_{x_p}$, the d_p are the diagonal elements of Matrix $D \in \mathbb{R}^{p \times p}$, and the loading Matrix V columns v_{x_p}.

PLS can also be applied as a supervised method by encoding Y as the class memberships. This will be closely related to Fisher Discriminant Analysis (Binary LDA).

2.2.9 Factor Analysis

The latent factors are latent variables that are not measured, but they are sometimes planned in the design of the measured dataset or reconstructed from the dataset after collection by grouping variables by their correlations. A group of observed variables will have a high correlation among themselves and a low correlation with other observed variables in other groups, such that each group will imply a different latent factor. The principal components capture the highest variance in the dataset in the first component and decreasing-order of variance in the following components. The main objective is to project a high-dimensional dataset into a smaller dimension model for visualisation or compression reasons. The PCA components are not interpretable as data objects themselves. The factors are interpretable data that cannot be measured directly in one variable, and several observed variables together represent one or more latent factor(s) (Brown, 2006).

Each measured indicator x_i contributes to the factors being measured η_j with loading or weight λ_{ij}, forming the general equation:

$x_i = \lambda_{i1}\eta_1 + \lambda_{i2}\eta_2 + \ldots + \lambda_{im}\eta_m + \varepsilon_i$

$x_i = \varepsilon_i + \sum_{j=0}^{m-1} \lambda_{ij}\eta_j$

A single equation that relates the observed variables x to the latent variables/factors η and the unique variance ε is: $x = W_x\eta + \varepsilon$.

That can be captured in matrix form as $\Sigma = W_x \Psi W_x^T + \Theta_\epsilon$, where $\Sigma \in \mathbb{R}^{d\times d}$ is the symmetric correlation matrix of d indicators $X \in \mathbb{R}^d$, $W_x \in \mathbb{R}^{d\times m}$ is the factor loadings matrix containing m $\lambda \in \mathbb{R}^d$, $\Psi \in \mathbb{R}^{m\times m}$ is the symmetric correlation matrix of the factor correlations, and $\Theta_\varepsilon \in \mathbb{R}^{d\times d}$ is the diagonal matrix of unique variances ε.

Various algorithms can estimate the factors and their loading. The most predominantly used are maximum likelihood (ML is most suitable to normally distributed datasets) and principal factors (PF makes no data distribution assumptions). Other methods include but are not limited to: weighted least squares, unweights least squares, generalised least squares, imaging analysis, minimum residual analysis, and alpha factoring (Brown, 2006).

The diagonalisation of the covariance or correlation matrix always calculates all the components at once. Also, the NIPALS method (Non-linear Iterative Partial Least Squares) explained in the previous section, calculates the components stepwise and is much faster than diagonalisation of the covariance matrix if only the first few eigenvalues are desired.

Also, Probabilistic PCA (PPCA) also estimates W and Ψ iteratively, using the Expectation.Maximisation (EM) algorithm (Burges, 2009).

Finding common factors in a dataset is often conducted in the exploratory model, in which the Exploratory Factor Analysis (EFA) method estimate both the number of factors and the loadings. EFA is like PCA, which is simpler to compute, accounting for the dataset's variance. EFA differ in considering the unique variance ε_i and explains the correlations among the observed variables while accounting for measurement errors. EFA can also be used to reconstruct the intercorrelations between the measured variables and a small set of latent factors.

Real-Life Application: a single factor of depression cannot be measured directly. Observed indicators, for example, can be scores for hopelessness, feelings of worthlessness/guilt, psychomotor retardation, and sleep disturbance. This four-dimensional dataset X can measure how they contribute to the depression latent factor η_1 as follows:

$x_1 = \lambda_{11} * \eta_1 + \varepsilon_1$

$x_2 = \lambda_{21} * \eta_1 + \varepsilon_2$

$x_3 = \lambda_{31} * \eta_1 + \varepsilon_3$

$x_4 = \lambda_{41} * \eta_1 + \varepsilon_4$

Check the Python code FA.ipynb for another psychometric dataset FA example.

2.2.10 Non-negative matrix factorisation (NMF)

NMF extracts a set of sparse meaningful factors from dataset X of non-negative values. It is a probabilistic model suitable for high-dimensional data. X$\in \mathbb{R}^{n \times d}$ is decomposed into W$\in \mathbb{R}^{n \times r}$, and H $\in \mathbb{R}^{r \times d}$ matrices with reduced dimensionality to rank r approximation. Matrix W columns contain the basis features; basis features are persistent features in all the n data points. Matrix H columns explain wherein X (coordinates) these bases are found and how important they are to help reconstruct the dataset. In facial images X dataset of n images and d pixels, the basis might be noses, eyes, hair, moustache, and other facial features. In text dataset X of n document and d words, the basis W can be topics that are defined to contain specific words, and H can be the importance of a word in the document. The rank r optimisation works by: $\underset{W \in \mathbb{R}^{n \times r}, H \in \mathbb{R}^{r \times d}}{\operatorname{argmin}} \|X - WH\|_f^2$ subject to W ≥ 0 and H ≥ 0. The partial derivatives are:

$$\Delta_W F = WHH^T - XH^T \geq 0, \text{ such that } W \circ \Delta_W F = 0, \text{ and } W \geq 0$$

$$\Delta_H F = W^T WH - W^T X \geq 0, \text{ such that } H \circ \Delta_H F = 0, \text{ and H} \geq 0$$

where ∘ is the component-wise product of two matrices.

The following generally are the steps performed to estimate W and H alternatively, such as the Hierarchical alternating least squares (HALS) algorithm:

Step 1: Initialize W and H randomly

Step 2: Repeat For ℓ = 1, 2, . . . , r:

1. Update W(:, ℓ) such that the objective function decreases: $\mathrm{W}(:,\ell) \leftarrow \underset{\mathrm{W}(:,\ell)\geq 0}{\operatorname{argmin}} \|X - \sum_{k\neq \ell} W(:,k)H(k,:) - W(:,\ell)H(\ell,:)\|_f$,
2. Update H accordingly to continue decreasing the objective function while maintaining the constraints:
3. Convergence is checked by: $C(W,H) = C_w(W) + C_H(H)$, where $C_w(W) = \|\min(W,0)\|_f + \|\min(\Delta_W F,0)\|_f + \|W \circ \Delta_W F\|_f$ for all constraints, and $C_H(H) = \|\min(H,0)\|_f + \|\min(\Delta_H F,0)\|_f + \|H \circ \Delta_H F\|_f$

Some regularisation is required to guarantee convergence. Detailed problems and research outcomes on how to avoid them can be found in (Suykens *et al.*, 2014).

Check the Python code NMF.ipynb for text mining NMF example compared with MiniBatchNMF and Latent Dirichlet Allocation (LDA) as implemented by scikit-learn.

2.2.11 Other Factorisations

The literature presents a plethora of algorithms that serve different objectives and constraints, and researchers will keep advancing the performance by addressing existing problems. Some of the factorisation techniques that we did not discuss are as follows:

- LU factorization: A = LU;
  - where L is the lower triangle and U is the upper triangle of matrix A. In the Gaussian Elimination algorithm used in chapter one to solve a system of equations Ax = b, we applied elementary row operations (ERO) to reduce a matrix to an upper triangular matrix. This is composed of a forward substitution Lz = b, and backward substitution, Ux = b.
- LU factorization with row pivoting: PA = LU;
- The Cholesky factorisation, $A = LL^T$
- The QR factorisation, $A \in \mathbb{R}^{m\times n}$= QR;

where Q ∈ $\mathbb{R}^{m\times m}$ has the special property of being an orthonormal matrix with respect to its columns, such that QTQ = I and R ∈ $\mathbb{R}^{m\times n}$ is an upper triangular matrix. If m > n, a reduced QR decomposition – or thin QR factorisation – can be computed, i.e. Q ∈ $\mathbb{R}^{m\times n}$ and R ∈ $\mathbb{R}^{n\times n}$.

- When a matrix is indefinite symmetric, there is a factorisation called the LDL^T (pronounce as L D L transpose) factorisation: A = LDL^T ;
  - where L is a unit lower triangular and D is diagonal. You may want to see if you can modify the derivation of the Cholesky factorisation to yield an algorithm for the LDL^T factorisation (Geijn and Quintana-Ort´, 2008).
- Dictionary Learning has many algorithms and implemented in sklearn.decomposition.DictionaryLearning.

2.2.12 Comparing Methods

You can notice that eigendecomposition Av = λBv is the fundamental decomposition technique that the most discussed algorithm reduces to, after considering the problem objectives and constraints. Table 1 is adopted from (Lu, Plataniotis and Venetsanopoulos, 2014) with the extra methods added and a summary of objectives and limitations of methods.

Table 1: Comparison of Linear Subspace Learning Methods

Method	Maximise	A	B	v	Objective	Limitations
PCA	Total scatter (variation)	S_T	I	u_1	only for the Gaussian data representation method. Closed Formula Exists.	First and second-order statistics: mean and std, ignoring correlations of higher statistics. Assumes linear dataset. Hard to interpret the PCs' relation to data – not suitable for feature extraction.
SVD	Covariance matrix	A^TA	U	V	Relates to data well. Singular values are stable for all matrices. Useful for compression.	

LDA	Between-class to within-class scatter ratio	S_B	S_W	u_1	Supervised Data Disciminitive method.	Suffer from high variance in SSS.
CCA	Correlation	$\begin{bmatrix} 0 & \Sigma_{xy} \\ \Sigma_{yx} & 0 \end{bmatrix}$	$\begin{bmatrix} \Sigma_{xx} & 0 \\ 0 & \Sigma_{yy} \end{bmatrix}$	$\begin{bmatrix} u_{x1} \\ u_{y1} \end{bmatrix}$	Correlation between two symmetric datasets	Require the inverse of Σ_{xx}, Σ_{yy}, which can be solved by regularisation or Cholesky decomposition. Suffer from high variance in SSS.
PLS	Covariance	$\begin{bmatrix} 0 & \Sigma_{xy} \\ \Sigma_{yx} & 0 \end{bmatrix}$	$\begin{bmatrix} I & 0 \\ 0 & \mathrm{I} \end{bmatrix}$	$\begin{bmatrix} u_{x1} \\ u_{y1} \end{bmatrix}$	Covariance between asymmetric Datasets Work well when Σ_{xx}, Σ_{yy} are singular.	Work well for a small number of latent variables.
FA	PLS and other variations can perform FA	$\begin{bmatrix} 0 & \Sigma_{xy} \\ \Sigma_{yx} & 0 \end{bmatrix}$	$\begin{bmatrix} I & 0 \\ 0 & \mathrm{I} \end{bmatrix}$	$\begin{bmatrix} u_{x1} \\ u_{y1} \end{bmatrix}$	Map Directly to meaningful unobserved data	
ICA	Iterative Algorithm				Works for non-Gaussian data. Higher-order statistics: kurtosis or negentropy. Iterative methods.	ICs are not ordered, and the actual number can not be estimated.
NMF	Iterative Algorithm				Sparse Meaningful features in high-dimensional datasets	Probabilistic model.

Since the Eigendecomposition is at the heart of dimensionality reduction methods, approximating this decomposition for a submatrix of the given dataset can speed up the training. The generalised Nyström method does this by a quadrature method that solves an integral equation by replacing the integral with a representative weighted sum. For a dataset $X \in \mathbb{R}^d$, the density p(x), is represented by the integral form, which reduces to a quadrature rule as follows:

$$\int k(x,y)u(y)p(y)dy = \lambda u(x) \approx \frac{1}{m}\sum_{i=1}^{m} k(x,x_i)u(x_i)$$

When the approximate equation on the right-hand side is applied to the sample points, it becomes a matrix equation $k_{mm}u_m = m\lambda u_m$, where u is ($u_i \equiv u(x_i)$) is the eigenfunction for some chosen m points such that the rank r of X (number of linearly independent columns) is $\gg m$, and k_{mm} means indices mm and that $K \in M_m$ is a submatrix of X, (with components $K_{ij} \equiv k(x_i,x_j)$). When m=r, then the approximation is exact. The eigenvalues λ_i approximately scale with the number of points chosen m. The full steps of solving the Nyström method exact and approximate variant (dropping the requirement that eigenvectors be orthogonal to each other) are explained in (Burges, 2009). The point to emphasise is that the projections of a mapped test point along principal components in a kernel feature space (the submatrix K) are equal to the expression for the approximate eigenfunctions evaluated at the new point, computed according to the approximate quadrature rule equation mentioned above. This mapping to a submatrix approximates the full eigenfunctions at all points. The main objective of the following section is to find Manifolds of lower dimensions that approximate the points in a dataset in the higher dimension.

2.3 Manifolds

Data in high dimensional spaces tend to be sparse, with most vectors containing zeros. This makes most vectors orthogonal to each other and equidistant, which means there is no structure to learn. These properties disable distance-measure-based dimensionality reduction algorithms to identify the linear subspace efficiently. Consequently, any regression or classification algorithm estimating the decision line or hyperplane in a higher dimension will fail to draw a line between equidistant points. Learning a Manifold M that contains the data points of the given dataset represented in a lower dimension than the input higher dimension space is a non-linear unsupervised dimensionality reduction approach that reduces the computation cost and increases the accuracy of the ML algorithm. In Natural Language Processing (NLP), words must be numerically encoded. If we encode all words in a dictionary, this might be sparse, and not all words will be used in a specific document or dataset. The word embeddings (manifold) from a given corpus can be learned using deep learning layers and can be constructed by statistical methods, maintaining closer codes for semantically close words.

This learning of the points' relationships to their neighbours in the manifold does not need to be using a similar relationship, equation or geometric shape. This would create a grid of some sort of basis along each dimension, creating a dimensionality curse. On the contrary, the manifold can be learned using several relationships in different neighbourhoods embedded in the manifold. For example, instead of learning the same unified principal components of a dataset and projecting the dataset onto the initial principal components for reduction, a manifold learning algorithm would partition the data into closer

neighbourhoods. Then, the algorithm would learn the principal components for each neighbourhood independently and project the relevant points onto their initial PCs that are different from the other neighbourhood, then form a manifold of all projections. Similar to the power method in estimating the Eigenvector, starting from random embedding and optimising it is a popular manifold learning approach. Check Python notebook Manifolds.ipynb for a comprehensive example using MNIST dataset comparing manifolds created by linear unsupervised PCA, supervised LDA and QDA, and manifold learning algorithms, MDS, Isomap, LLE, t-SNE, and Spectral Embedding.

2.3.1 Multidimensional Scaling (MDS)

MDS searches for a measure of dissimilarity between each pair of data points in X ∈ $\mathbb{R}^{m\times d}$ to map them to a low dimensional Euclidean Space, forming a squared distances matrix to be used for visualisation or dimensionality reduction and feature extraction. This can be performed using similar steps to the following:

Step 1: Assign some points to coordinates in n-dimensional space.

1. Choose N points to be 2D or 3D to be able to visualise and easily modelled. The coordinates' orientation is arbitrary and depends on the features' nature. For example, two coordinate axes representing north/south and east/west are suitable for modelling a dataset containing map locations.

Step 2: Calculate the symmetric similarity matrix A using a suitable distance measure for all pairs of points.

2. The Euclidean distance is the Euclidean norm ||. || that is based on the Pythagorean theorem becomes more complicated for n-dimensional space (see https://hlab.stanford.edu/brian/euclidean_distance_in.html). For two pairs of data points x, y ∈ $\mathbb{R}$: dist(x, y) = x-y, for x, y ∈ $\mathbb{R}^2$, dist(x, y) = $\sqrt{x^2 - y^2}$, and for x, y ∈ $\mathbb{R}^d$, dist(x, y)= $\sqrt{\sum_{i=1}^{d}(x_i^d + y_i^d)}$. This results in the similarity matrix A whose ij-th element is $\left\|x_i - x_j\right\|^2$ for some $x_i, x_j \in \mathbb{R}^d$. Other distance metrics can be used. Also, non-metric scaling in ordinal MDS can be performed by using a given rank ordering of the original data points in this matrix that is minimised using a suitable cost function of the difference between the embedded squared distances and some monotonic function of the dissimilarities.

Step 3: Find the lower-dimensional embedding:

3. The Schoenberg theorem states that the class of symmetric matrices A $\in \mathbb{R}^{n\times n}$ such that $A_{ij} \geq 0$ and $A_{ii} = 0$ $\forall$i, j. Then $\bar{A} \equiv -PAP$ is positive semi-definite if and only if A is a distance matrix (with embedding space $\mathbb{R}^d$ for some d). Given that A is a distance matrix, the minimal embedding dimension d is the rank of $\bar{A}$, and the embedding vectors are any set of Gram vectors of $\bar{A}$, scaled by a factor of $\frac{1}{\sqrt{2}}$.
   a. Defining the projection matrix $P^c \equiv (1 - ec')$, for any c$\in$R^m such that $e'c = 1$, then for any conditionally negative definite (CND) matrix A, the matrix $-P^cAP'^c$ is positive semi-definite (and hence a dot product matrix). We can map a distance matrix A to a dot product matrix K by using P^c in the above manner for any set of numbers c_i that sum to unity. Again, this reduces to Eigendecomposition $\bar{A}E = E\Lambda$, where $\bar{A}$ is the positive semi-definite matrix that we need to solve for, E be the matrix of column eigenvectors $e^{(\alpha)}$ (labelled by α), ordered by eigenvalue λ_α, so that the first column is the principal Eigenvector and Λ is the diagonal matrix of eigenvalues. Defining the matrix $\tilde{E} \equiv E\sqrt{\Lambda}$, we see that the Gram vectors are just the rows of $\tilde{E}$.
   b. If $\bar{A} \in \mathbb{R}^{n\times n}$ has rank r ≤ n, then the final n − r columns of $\tilde{E}$ will be zero, and we have directly found the r-dimensional embedding vectors that we are looking for. If $\bar{A} \in \mathbb{R}^{n\times n}$ is full rank, but the last n − p eigenvalues are much smaller than the first p, then it's reasonable to approximate the ith Gram vector by its first p components $\sqrt{\lambda_\alpha}\tilde{e}_\alpha^{(i)}$, α = 1, …, p, and we have found a low dimensional approximation. MDS, Laplacian eigenmaps, and spectral clustering perform this latter approach of removing the last few components. The unexplained squared residuals measure the quality of the approximation. Thus the fraction of the "unexplained residuals" is $\sum_{a=p+1}^{r} \frac{\lambda_a}{\sum_{a=1}^{r}\lambda_a}$, in analogy to the fraction of "unexplained variance" in PCA.

Step 4: Compare the similarity matrix with the original input matrix by evaluating the stress function.

4. Stress is a goodness-of-fit measure based on differences between predicted and actual distances. In the original MDS paper (Kruskal, 1964), it was mentioned that fits close to zero are excellent, while anything over 0.2 should be considered "poor". More recent authors suggest evaluating stress based on the quality of the distance matrix and how many objects are in that matrix.

Step 5: Adjust coordinates, if necessary, to minimise stress by repeating the above steps.

The complexity of MDS is reduced by using a Landmark MDS algorithm (Borg, Groenen and Mair, 2013).

2.3.2 Isometric Feature Map (Isomap)

MDS maps the dataset to a lower-dimensional embedding such that the data points are isotropically represented in the lower space (maintaining equivalent distances) without modelling the underlying manifold. Isomap and Locally Linear Embedding are methods to model the lower dimensional manifold without keeping the equivalent distances constraint. To preserve the non-linear structures in the dataset, Isomap assumes that the data points lie on a curve and not a straight line, and instead of measuring the dissimilarity matrix between data points, it measures the distance along the curve between the two points. It accounts only for large distances along this curve, even if the two points are close in $\mathbb{R}^d$. The basic idea is to construct a graph whose nodes are the data points, where a pair of nodes are adjacent only if the two points are close in $\mathbb{R}^d$, and then to approximate the geodesic distance along the manifold between any two points as the shortest path in the graph, computed using the Floyd algorithm or the faster Dijkstra's algorithm with Fibonacci heap (Qu and Cai, 2017); and finally to use MDS to extract the low dimensional representation (as vectors in $\mathbb{R}^{d'}, d' \ll d$) from the resulting matrix of squared distances. Isomap does not provide a direct mapping function: $\mathrm{I}: \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$.

2.3.3 Locally Linear Embedding (LLE)

LLE models the manifold by treating it as a union of linear patches, in analogy to using coordinate charts to parameterise a manifold in differential geometry. Suppose $X \in \mathbb{R}^{m \times d}$ with each point $x_i \in \mathbb{R}^d$ has a small number of close neighbours indexed by the set N(i) up to n neighbours, and let $y_i \in \mathbb{R}^{d'}$ be the low-dimensional representation of x_i. The idea is to express each x_i as a linear combination of its neighbours and then construct the y_i so that they can be expressed as the same linear combination of their corresponding neighbours (Ghojogh *et al.*, 2020). Similar steps to the following can achieve this:

Step 1: Find the W's $\in \mathbb{R}^{m \times n}$ that minimises the sum of the reconstruction errors

1. The reconstruction error: $E_i \equiv \left\| x_i - \sum_{j \in \mathrm{N(i)}} W_{i,j}\, x_j \right\|^2$ that is invariant of global translation, which creates the constraint: $\sum_{j \in \mathrm{N(i)}} W_{i,j} = 1 \; \forall i$. It should also be invariant of data scaling, rotation, and/or reflection.
2. This is achieved by minimising the following objective function:

$$F \equiv \sum_i F_i \equiv \sum_i \left(\frac{1}{2} \left\| x_i - \sum_{j \in \mathrm{N(i)}} W_{i,j}\, x_j \right\|^2 - \lambda_i \left(\sum_{j \in \mathrm{N(i)}} W_{i,j} - 1 \right) \right)$$

3. Differentiate each F_i with respect to $W_{i,j}$, and require that $W_{i,j}$vanishes, and set to zero.

Step 2: Find a set of eigenvectors $y_i \in \mathbb{R}^{d'}$

4. Perform Eigendecomposition of the product of two sparse matrices $(1 - W)'(1 - W) \in \mathbb{R}^m$ with the smallest eigenvalues guarantees that the y is zero mean (since they are orthogonal to e).
5. This is achieved by minimising the following objective function:

$$F = \sum_i \left(\frac{1}{2} \left\| y_i - \sum_j W_{i,j}\, y_j \right\|^2 - \frac{1}{2} \sum_{\alpha\beta} \lambda_{\alpha\beta} \left(\sum_i \frac{1}{m} Y_{i\alpha} Y_{i\beta} - \delta_{\alpha\beta} \right) \right)$$

6. Differentiate each F with respect to $Y_{k\delta}$ and choosing $\lambda_{\alpha\beta} = \lambda_\alpha \delta_{\alpha\beta} \equiv \Lambda_{\alpha\beta}$ gives the matrix equation: $(1 - W)'(1 - W)Y = \frac{1}{m} Y\Lambda$

LLE tend to create a dense manifold in the centre, with various emerging rays. This is justified by the LLEs approach to learning the multilinear function from a set of simpler linear functions, making it suitable for simple structures in smaller datasets.

2.3.4 t-distributed Stochastic Neighbor Embeddings (t-SNE)

t-SNE represents a dataset as a t-distribution or normal distribution of smaller sizes. It is almost like a neural network being trained using a stochastic gradient descent algorithm to minimise the entropy metric. It is an unsupervised approach that clusters that data points into local groups of neighbourhoods and focuses on learning about their local structures than unfolding them. It is an efficient manifold learning algorithm for multi-scale datasets with complex structures and multiple manifolds. However, it is computationally expensive compared to the other manifold learning algorithms.

2.3.5 Spectral Graph decomposition: Spectral Clustering

A simple mapping from each data point in a dataset to a node in an undirected graph G, and apply minimum spanning tree or nearest neighbour algorithm to fill the adjacency matrix

W_{ij}, with a similarity measure between node i and node j. The normalised Laplacian matrix for any weighted, undirected graph is defined by $L \equiv D^{-\frac{1}{2}}LD^{-\frac{1}{2}}$, where $L_{ij} \equiv D_{ij} - W_{ij}$ and $D_{ij} \equiv \delta_{ij}(\sum_k W_{ik})$. L is positive semi-definite, and so is the normalised Laplacian. The spectral Graph theory states that the graph spectrum properties defined by the eigenvalues of its Laplacian, characterise the global graph properties as follows:

- A complete graph (that is, one for which every node is adjacent to every other node) has a single zero eigenvalue, and all other eigenvalues are equal to $\frac{m}{m-1}$.
- If G is connected but not complete, its smallest non-zero Eigenvalue is bounded above by unity.
- The number of zero Eigenvalues is equal to the number of connected components in the graph, and in fact, the spectrum of a graph is the union of the spectra of its connected components.
- The sum of the eigenvalues is bounded above by m, with equality if G has no isolated nodes.

In light of these results, it seems reasonable to expect that global properties of the data — how it clusters or what dimension manifold it lies on — might be captured by properties of the Laplacian. Both Laplacian Eigenmaps and Spectral Clustering manifold learning methods apply spectral graph theory to model the lower-dimensional manifold a dataset lives in. Clustering generally reduces the dimensionality of the number of data points to the number of clusters of similar structural features at its largest scale. Partitioning a graph into two disjoint clusters of nodes requires removing arcs such that the cut is defined as the sum of removed arcs weights. The minimum cut implies the maximum dissimilarity between the clusters. Each node is labelled by $z_i = 1$ for nodes in one cluster and $z_i = -1$ for nodes in the other cluster. The solution to the normalised min-cut problem is given by: $\min_y \frac{y'Ly}{y'Dy}$ such that $y_i \in \{1, -b\}$ and $y'De = 0$, where $y \equiv (e + z) + b(e - z)$, and b is a constant that depends on the partition. The following steps are generally followed:

Step 1: Weighted Graph Construction.

1. Transform the raw input data into graph representation using affinity (adjacency) matrix representation A.

Step 2: Graph Laplacian Construction.

2. Unnormalised Graph Laplacian is constructed as L = D - A for and normalised one as $L \equiv D^{-\frac{1}{2}}(D - A)D^{-\frac{1}{2}}$

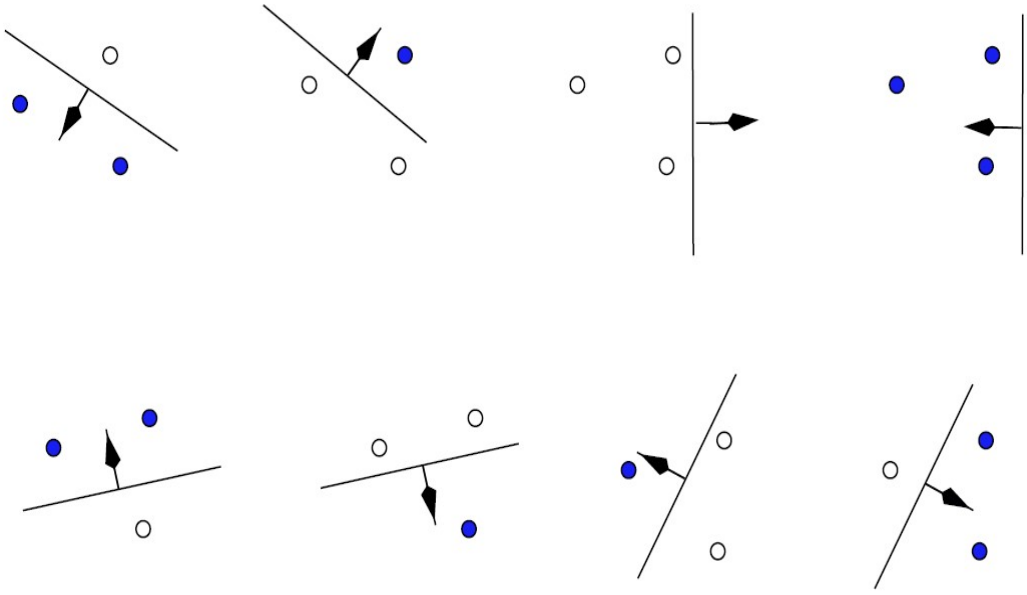
Step 3: Partial Eigenvalue Decomposition.

3. Eigenvalue decomposition is done on graph Laplacian: $Ly = \lambda Dy$
4. The clustering is achieved by thresholding a single eigenvector y_i so that the nodes are split into two disjoint sets. The dimension reduction is achieved by treating the element y_i as the first component of a reduced dimension representation of sample x_i. Otherwise, a simple clustering algorithm such as K-means can be applied to the reduced dimensionality dataset and is observed to perform much better than on the full dataset.

2.4 Mapping to the Higher Dimensions

The estimated decision line/hyperplane function needs to shatter the dataset's N points, i.e. the dataset is separable by the classifier. Figure 3a) illustrates the classifications of N=3 points $\in \mathbb{R}^2$ classified into blue (1) and white (-1) classes. All possible assignments of labels (all random datasets that can be obtained) are 2^3=8 different possible datasets. The figure shows the estimated decision line for all possible datasets as a set of functions such as f(x, w) to separate the classes. In $\mathbb{R}^2$ it is possible to find three points that this set of functions can shatter, but it is not possible to find four. This is measured by the Vapnik Chervonenkis (VC) dimension as defined below. The set of oriented lines in $\mathbb{R}^2$ is three. Figure 3b) shows how a given set of N=4 points $\in \mathbb{R}^1$ are not shattered by the function f(w, x) = sine (wx), which generally has an infinite VC dimension for being able to shatter a subset of the set $\{2^{-m} | m \in \mathbb{N}\}$. This is because these points are equally spaced and labelled in this order.

a)



b)

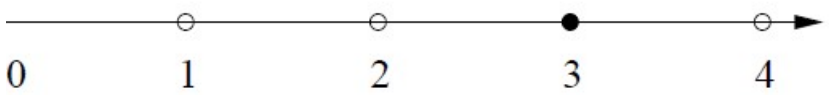


Figure 3a) Three points in $\mathbb{R}^2$ that shattered by decision lines function.b) Four points that cannot be shattered by {sin(wx)}, despite infinite VC dimension by the function set.

2.4.1 Support Vector Machine (SVM)

SVM estimates the decision line using a convex quadratic programming algorithm. Linear programming minimises or maximises a linear function subject to defined constraints, such as solving for minimum weights to satisfy the classification or regression of a given dataset, given regularisation constraints and others. A quadratic programming optimisation problem is a form of non-linear programming involving a quadratic convex objective function, and the points that satisfy the required constraints also form a convex set. Any linear constraint defines a convex set, and a set of N simultaneous linear constraints defines the intersection of N convex sets, which is also a convex set. For a labelled dataset $\{x_i, y_i\}$ i from 1 to N, $y \in \{1, -1\}$, and $x_i \in \mathbb{R}^d$, the optimisation aims to find the weights/parameters for the separating hyperplane that separates the positive classes from the negative classes. For the linearly separable dataset, a possible set of steps are as follows:

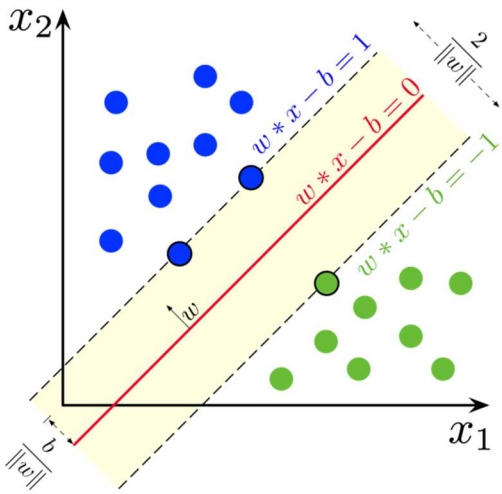


Figure 4: Maximum-margin hyperplane and margin for an SVM trained on two classes with $x \in \mathbb{R}^2$. Samples on margins are called support vectors.(Larhmam, 2018)

Step 1: the geometric preliminaries of the algorithm to identify the optimisation requirements:

1. Generally, any data point x which lies on the hyperplane satisfies wx + b = 0, where w is normal to the hyperplane, $\frac{|b|}{\|w\|}$ is the perpendicular distance from the hyperplane to the origin, and $\|w\|$ is the Euclidean norm of w. But in SVM, we need to find the hyperplane that maximises the margin between the closest points from both classes. Therefore, no x points should fall on the separating hyperplane.
2. To achieve this, we need to identify two hyperplanes, each one passes through the support vectors (points x closest to the hyperplane from both directions). Therefore, all data points support the following constraints:
    a. $wx_i + b \geq +1$ for $y_i = +1$, these points lie on the hyperplane H_1: $wx_i + b = 1$ with normal w, and perpendicular distance from the origin = $\frac{|1-b|}{\|w\|}$
    b. $wx_i + b \leq -1$ for $y_i = -1$, these points lie on the hyperplane H_2: $wx_i + b = -1$ with normal w, and perpendicular distance from the origin = $\frac{|-1-b|}{\|w\|}$
3. Let d_+ (d_-) be the shortest distance from the separating hyperplane to the closest positive (negative) example, such that the margin of the separating hyperplane is equal to $d_+ + d_-$. No data points fall between H_1 and H_2.
4. To maximise the margin between H_1 and H_2 hyperplanes, we need to minimise $\|w\|^2$ subject to the combined constraints above as: $y_i (wx_i + b) - 1 \geq 0 \; \forall i$, such that $d+ = d- = \frac{1}{\|w\|}$, and the margin between H_1 and H_2 = $\frac{2}{\|w\|}$.

Step 2: Apply the Lagrangian optimisation formulation

5. The Primal Lagrangian equation adds a Lagrange constant for every data point, checking if it is a support vector $\alpha_i > 0$, or not $\alpha_i = 0$ if it lies on one of the hyperplanes. The equation becomes: $L_P = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{N} \alpha_i y_i \, (x_i . w + b) + \sum_{i=1}^{N} \alpha_i$
6. We have two sets of constraints forming a Wolfe dual problem:
    a. Minimise L_P with respect to w and b, subject to the constraint that the derivatives of L_P with respect to all the α_i vanish, all subject to the constraints $\alpha_i \geq 0$.:
    b. Maximise L_P with respect to α_i, subject to the constraints that the gradient of L_P with respect to w and b vanish, and subject also to the constraints that the $\alpha_i \geq 0$: $w = \sum_{i=1}^{N} \alpha_i y_i \, x_i$
    c. A mild constraint to make b=0, will make all hyperplanes pass through the origin and reduce the degree of freedom by one: $\sum_{i=1}^{N} \alpha_i y_i = 0$
    d. Now the Dual Lagrangian equation is: $L_D = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \, x_i . x_j$ subject to the mild constraint in 6.c. by adding an extra Lagrange constant λ.
    e. We can use the Hessian $H_{ij} = y_i y_j x_i \, . \, x_j$, to formulate the Dual Lagrangian equation is: $L_D = \sum_{i=1}^{N+1} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N+1} \alpha_i H_{ij} \alpha_j - \lambda \sum_{i=1}^{N+1} \alpha_i y_i$
7. Differentiate with respect to w and b:
    a. $\frac{\delta L_D}{\delta \alpha_i} = (H\alpha)_i + \lambda y_i = 1 \; \forall i$ = 1, … l , one for each Lagrangian constraint.
    b. $\frac{\delta L_P}{\delta w_v} = w_v - \sum_i \alpha_i y_i x_{iv} = 0$ for v=1, …d
    c. $\frac{\delta L_P}{\delta b} = -\sum_i \alpha_i y_i = 0$

Toy examples of the linearly separable data are “AND”, and “OR” are shown in Figure 5. The “XOR” dataset is not linearly separable. One solution requires two separation 1-D lines processed in 2 layers, and one class spans two areas. Another solution presented in Figure 6 is to map the dataset to a higher dimension, in which it will be separable linearly; for example, a 2-D plane separates the “XOR” dataset after mapping it to $\mathbb{R}^3$ using the following mapping function:

$\Phi(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) = \begin{bmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix}$, such that:

$\Phi(0, 0) = (0, 0, 0)$
$\Phi(0, 1) = (0, 1, 0)$
$\Phi(1, 0) = (1, 0, 0)$
$\Phi(1, 1) = (1, 1, 1)$

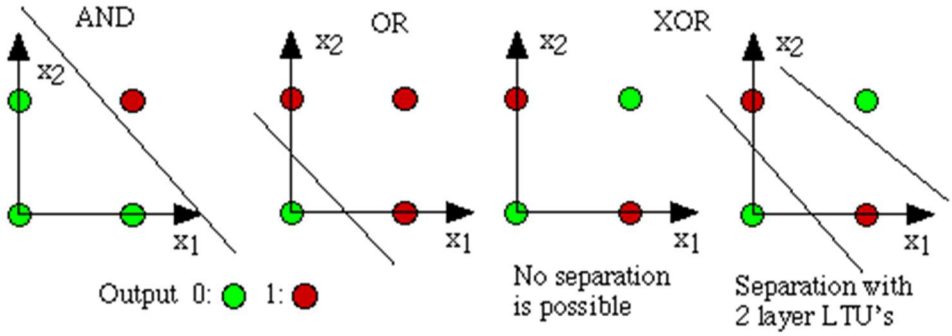


Figure 5: Linearly Separable toy examples in the AND and OR functions. The XOR function is not linearly separable and requires 2 1-D lines processed in 2 Layers

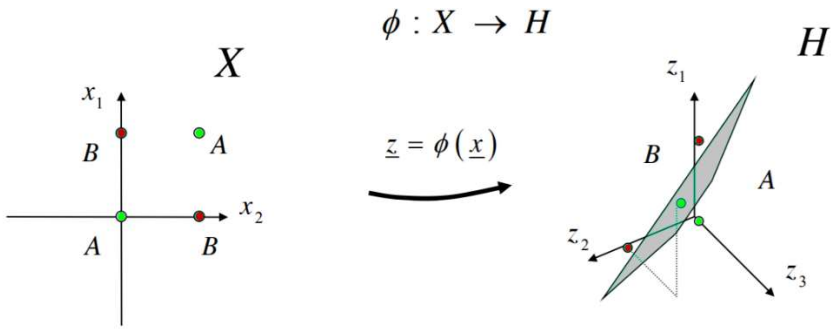


Figure 6: Mapping the XOR dataset to a higher dimension here $\mathbb{R}^3$, enable it to be linearly separable by 2-D plane as shown

2.4.2 Kernel Trick

The following concepts need to be understood first about the measure of data separability given classifier functions in order to understand the non-linearly separable dataset: VC Dimension, and Hilbert Spaces, in which a non-linear dataset in the low dimension can be separable in the higher dimension.

The Vapnik Chervonenkis (VC) dimension is a non-negative integer property that measures the ML algorithm flexibility to learn a decision line for a large set of points of any possible labelling. The ML algorithm produces a set of functions H= {f(w)} (w is the estimated vector of weights/parameters for different datasets and not a specific value for w that makes a function a particular function for a particular dataset). It can be defined for various classes of function f (different algorithms such as different complexity, higher polynomial or radial basis functions). Here we will only consider functions that correspond to the two-class pattern recognition case so that f(x, w) ∈ {1, -1} $\forall x, w$. A given set C of N points labelled in all possible 2^N ways are said to be shattered by the ML algorithm when for each labelling, a member of the set {f(w)} can be found that correctly assigns those labels. The intersection of

sets H and C is defined as the following set family: $H \cap C = \{h \cap C : h \cap H\}$. C is shattered by H, if $H \cap C$ contains all subsets of C: $|H \cap C| = 2^{|C|}$. The VC dimension for the set of functions H is defined as the largest cardinality of sets (training points) that can be shattered by it (correctly classified without errors). VC dimension is ∞ if it is too large. Note that if the VC dimension is h, then there exists at least one set of h points that can be shattered, but in general, it will not be true that every set of h points can be shattered (Burges, 1998). The VC dimension of a finite classification model, which can return at most 2^d different classifiers, is at most d, depending on the structure of the model (chosen class of the function).
Generally, some bounds are proven on the different dimensionalities and classifiers:

- A constant classifier (with no parameters), its VC dimension is 0 since it cannot shatter even a single point.
- A single-parametric threshold classifier on real numbers; i.e., for a certain threshold w, the classifier f(w) returns one if the input number is larger than w and 0; otherwise, the VC dimension of f(w) is one because: it can shatter a single point, but it cannot shatter any set of two points.
- A single-parametric interval classifier on real numbers; i.e., for a certain threshold w, the classifier f(w) returns one if the input number is within the interval w and 0; otherwise, then the VC dimension of f(w) is two because: it can shatter some sets of two points, but it cannot shatter any set of three points.
- In a line classifier model (such as regression and perceptron), for each pair of distinct points, there is one line that contains both of them, lines that contain only one of them, and lines that contain none of them, so every set of size two is shattered. The line should separate positive data points from negative data points. There exist sets of 3 points that can indeed be shattered using this model (any 3 points that are not collinear can be shattered). However, no set of 4 points can be shattered because they can be partitioned into two subsets with intersecting convex hulls, so it is impossible to separate one of these subsets from the other. Thus, the VC dimension of this particular classifier is 3. This is the example shown in Figure 3a).
- A k-nearest neighbour with a k=1 classifier has infinite VC dimension and zero prediction error as, eventually, all points will be learned.
- A single-parametric sine classifier on real numbers; i.e., for a certain threshold w, the classifier f(w) returns one if the input number x has sine (wx) > 0 and 0 otherwise, then the VC dimension of f(w) is infinite because: it can shatter a finite subset of the set $\{2^{-m} | \ m \in \mathbb{N}\}$. Figure 3b) shows an example of a set of

points that are not shattered by this function to emphasise that not every set of cardinality equal to the VC dimension of a classifier will be shattered.

Hilbert Space $\mathcal{H}$ is a generalisation of the Euclidian space in the infinite dimension. More details will be explained in chapter five. In the infinite dimension, the inner product in step 6.d in the SVM linear solution steps above, $(\mathrm{x}_i.\,\mathrm{x}_j)$ can be performed after mapping both vectors to the higher dimensions using a mapping function: Φ: $\mathbb{R}^d \rightarrow \mathcal{H}$, where $\mathcal{H}$is an infinite dimension and $\mathbb{R}^d$ is considered in the $\mathcal{L}$ as a lower dimension. The algorithm would do the dot products as $(\Phi(\mathrm{x}_i).\,\Phi(\mathrm{x}_j))$ everywhere a lower-dimensional dot product $(\mathrm{x}_i.\,\mathrm{x}_j)$ is applied. Possible mappings to $\mathbb{R}^3$ are as follows:

$$\Phi(\begin{bmatrix}\mathrm{x}_1\\ \mathrm{x}_2\end{bmatrix}) = \begin{bmatrix}\mathrm{x}_1^2\\ \sqrt{2}\mathrm{x}_1\mathrm{x}_2\\ \mathrm{x}_2^2\end{bmatrix}, \text{ or } \Phi(\begin{bmatrix}\mathrm{x}_1\\ \mathrm{x}_2\end{bmatrix}) = \frac{1}{\sqrt{2}}\begin{bmatrix}(\mathrm{x}_1^2-\mathrm{x}_2^2)\\ 2\mathrm{x}_1\mathrm{x}_2\\ (\mathrm{x}_1^2+\mathrm{x}_2^2)\end{bmatrix}$$

Or a possible mapping to $\mathbb{R}^4$:

$$\Phi\left(\begin{bmatrix}\mathrm{x}_1\\ \mathrm{x}_2\end{bmatrix}\right) = \begin{bmatrix}\mathrm{x}_1^2\\ \mathrm{x}_1\mathrm{x}_2\\ \mathrm{x}_1\mathrm{x}_2\\ \mathrm{x}_2^2\end{bmatrix}$$

We can replace the mapping with a Kernel function $\mathrm{K}(\mathrm{x}_i, \mathrm{x}_j) = (\Phi(\mathrm{x}_i).\,\Phi(\mathrm{x}_j))$ that produces the same result without explicitly mapping every vector in the dataset or even defining a specific mapping. This will be further explained in chapter five while discussing the representer theorem and how it reduces searching the large space $\mathcal{H}$ to just finding the optimal values of the m coefficients $\alpha_1, \ldots, \alpha_m$ of $\mathrm{x}_1, \ldots, \mathrm{x}_m$ with each x_i a vector of the features. Example Kernel functions are:

- $\mathrm{K}(\mathrm{x}_i, \mathrm{x}_j) = (\mathrm{x}_i.\,\mathrm{x}_j + 1)^p$, results in a classifier polynomial equation of degree = p.
- $\mathrm{K}(\mathrm{x}_i, \mathrm{x}_j) = e^{\frac{-\|\mathrm{x}_i-\mathrm{x}_j\|^2}{2\sigma^2}}$, results in a classifier Gaussian radial basis function (RBF) equation.
- $\mathrm{K}(\mathrm{x}_i, \mathrm{x}_j) = \tanh(\kappa\mathrm{x}_i.\,\mathrm{x}_j - \delta)$, results in a particular kind of two-layer sigmoidal neural network.

This requires that the Hilbert Space Higher dimension $\mathcal{H}$ defines an inner product (not just the scalar dot product) such that any Cauchy sequence of points (that grows very close as the sequence progresses less than a given small positive distance) converges to a point in the space.

Mercer's condition: The kind of Kernel functions that define a dot product in some infinite space should conform with Mercer's condition, which states that there exists a mapping Φ and an expansion such that:

$\mathrm{K}(x,\mathrm{y}) = \sum_i \Phi(\mathrm{x})_i \Phi(\mathrm{y})_i$ if and only if, for any g(x) such that $\int g(x)^2\, dx$ is finite. Then, $\int \mathrm{K}(x,\mathrm{y}) g(x) g(y)\, dxdy \geq 0$. Therefore, any kernel satisfies Mercer's condition if it is of the form $\mathrm{K}(x,\mathrm{y}) = \sum_{p=0}^{\infty} c_p\, (x.y)^p$, where the c_p are positive real coefficients, and the series is uniformly convergent. Using such kernel produces a Hilbert Space Higher dimension $\mathcal{H}$ of the dimension= the combination $C(d+p-1,p) = C\binom{d+p-1}{p} = \frac{(d+p-1)!}{(p!(d+p-1)!)}$. For example, for a degree p = 4 polynomial, and for images data consisting of 16 by 16 pixels (d=256), dim($\mathcal{H}$) is 183,181,376.

Python notebook Classification_Linear_NonLinear.ipynb compares a number of classification algorithms on different types of datasets. It shows clearly the linearly separable datasets and the non-linearly separable ones classified by different SVM kernel functions and other linear and non-linear algorithms.

References

Borg, I., Groenen, P.J.F. and Mair, P. (2013) *Applied multidimensional scaling*. Berlin London: Springer (SpringerBriefs in statistics).

Brown, T.A. (2006) *Confirmatory factor analysis for applied research*. New York: Guilford Press (Methodology in the social sciences).

Burges, C.J.C. (1998) 'A Tutorial on Support Vector Machines for Pattern Recognition', *SUPPORT VECTOR MACHINES*, 2, pp. 121–167.

Burges, C.J.C. (2009) 'Dimension Reduction: A Guided Tour', *Foundations and Trends® in Machine Learning*, 2(4), pp. 275–364. Available at: https://doi.org/10.1561/2200000002.

Carter, T.A. (1995) *Linear Algebra, An Introduction to Linear Algebra for Pre-Calculus Students*. Rice University.

Deisenroth, M.P., Faisal, A.A. and Ong, C.S. (2019) *Mathematics for Machine Learning*. Cambridge University Press.

Downey, A. (2013) *Think Bayes*. First edition. Sebastopol, CA: O'Reilly.

Farrell, P. (2020) *The statistics and calculus workshop a comprehensive introduction to mathematics in Python for artificial intelligence applications*. Available at:

http://www.vlebooks.com/vleweb/product/openreader?id=none&isbn=9781800208360 (Accessed: 25 October 2021).

Geijn, R. van de (2012) *ULAFF: Linear Algebra: Foundations to Frontiers*. Available at: http://www.ulaff.net/ (Accessed: 6 March 2021).

Geijn, R.A. van de and Quintana-Ort´, E.S. (2008) *The Science of Programming Matrix Computations*. www.lulu.com. Available at: http://z.cs.utexas.edu/wiki/LA.wiki/books/TSoPMC/.

Ghojogh, B. *et al.* (2020) 'Locally Linear Embedding and its Variants: Tutorial and Survey'. arXiv. Available at: http://arxiv.org/abs/2011.10925 (Accessed: 6 September 2022).

Kruskal, J.B. (1964) 'Nonmetric multidimensional scaling : a numerical method.', 29(b), pp. 115–130.

Larhmam (2018) *Maximum-margin hyperplane and margin for an SVM trained on two classes. Samples on margins are called support vectors.* Available at: https://commons.wikimedia.org/wiki/File:SVM_margin.png#metadata (Accessed: 2 November 2021).

Leskovec, J., Rajaraman, A. and Ullman, J.D. (2014) *Mining of Massive Datasets*. Second edition. Cambridge: Cambridge University Press.

Lu, H., Plataniotis, K.N. and Venetsanopoulos, A.N. (2011) 'A survey of multilinear subspace learning for tensor data', *Pattern Recognition*, 44(7), pp. 1540–1551. Available at: https://doi.org/10.1016/j.patcog.2011.01.004.

Lu, H., Plataniotis, K.N. and Venetsanopoulos, A.N. (2014) *Multilinear subspace learning: dimensionality reduction of multidimensional data*. Boca Raton, Florida: CRC Press/Taylor & Francis Group (Chapman & Hall/CRC machine learning & pattern recognition series).

Qu, T. and Cai, Z. (2017) 'An improved Isomap method for manifold learning', *International Journal of Intelligent Computing and Cybernetics*, 10(1), pp. 30–40. Available at: https://doi.org/10.1108/IJICC-03-2016-0014.

Sivia, D.S. and Skilling, J. (2006) *Data analysis: a Bayesian tutorial*. 2nd ed. Oxford ; New York: Oxford University Press (Oxford science publications).

Suykens, J.A.K. *et al.* (eds) (2014) *Regularization, optimization, kernels, and support vector machines*. *ROKS*, Boca Raton London New York: CRC Press, a Chapman & Hall book (Chapman & Hall/CRC machine learning & pattern recognition series).

VON HILGERS, P. and LANGVILLE†, A.N. (no date) 'THE FIVE GREATEST APPLICATIONS OF MARKOV CHAINS'. Available at: http://langvillea.people.cofc.edu/MCapps7.pdf.