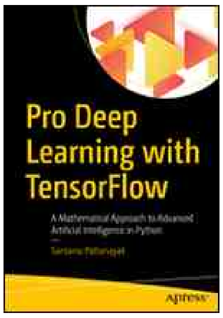


Pro Deep Learning with TensorFlow: A Mathematical Approach to Advanced Artificial Intelligence in Python

by Santanu Pattanayak

Apress. (c) 2017. Copying Prohibited.

Chapter 1: Mathematical Foundations

© Santanu Pattanayak 2017
S. Pattanayak, *Pro Deep Learning with TensorFlow*, https://doi.org/10.1007/978-1-4842-3096-1_1

Overview

Deep learning is a branch of machine learning that uses many layers of artificial neurons stacked one on top of the other for identifying complex features within the input data and solving complex real-world problems. It can be used for both supervised and unsupervised machine-learning tasks. Deep learning is currently used in areas such as computer vision, video analytics, pattern recognition, anomaly detection, text processing, sentiment analysis, and recommender system, among other things. Also, it has widespread use in robotics, self-driving car mechanisms, and in artificial intelligence systems in general.

Mathematics is at the heart of any machine-learning algorithm. A strong grasp of the core concepts of mathematics goes a long way in enabling one to select the right algorithms for a specific machine-learning problem, keeping in mind the end objectives. Also, it enables one to tune machine-learning/deep-learning models better and understand what might be the possible reasons for an algorithm's not performing as desired. Deep learning being a branch of machine learning demands as much expertise in mathematics, if not more, than that required for other machine-learning tasks. Mathematics as a subject is vast, but there are a few specific topics that machine-learning or deep-learning professionals and/or enthusiasts should be aware of to extract the most out of this wonderful domain of machine learning, deep learning, and artificial intelligence. Illustrated in Figure 1-1 are the different branches of mathematics along with their importance in the field of machine learning and deep learning. We will discuss the relevant concepts in each of the following branches in this chapter:

- Linear algebra
- Probability and statistics
- Calculus
- Optimization and formulation of machine-learning algorithms

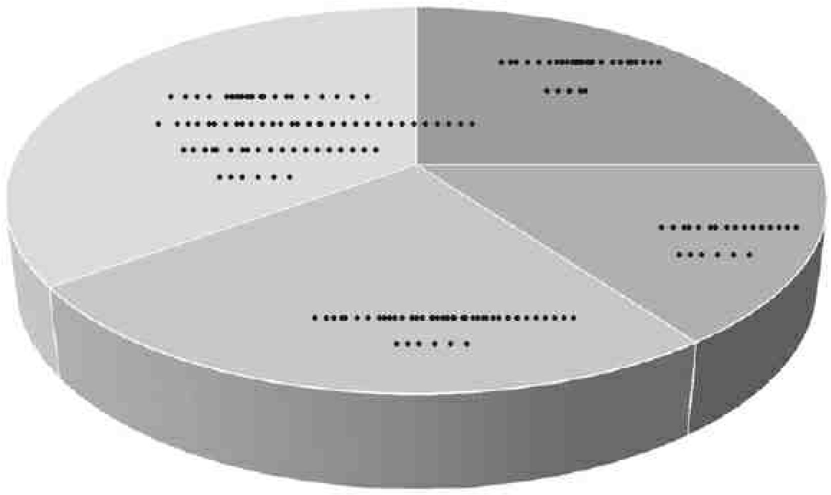


Figure 1-1: Importance of mathematics topics for machine learning and data science

Note Readers who are already familiar with these topics can chose to skip this chapter or have a casual glance through the content.

Linear Algebra

Linear algebra is a branch of mathematics that deals with vectors and their transformation from one vector space to another vector space. Since in machine learning and deep learning we deal with multidimensional data and their manipulation, linear algebra plays a crucial role in almost every machine-learning and deep-learning algorithm. Illustrated in Figure 1-2 is a three-dimensional vector space where v_1, v_2 and v_3 are vectors and P is a 2-D plane within the three-dimensional vector space.

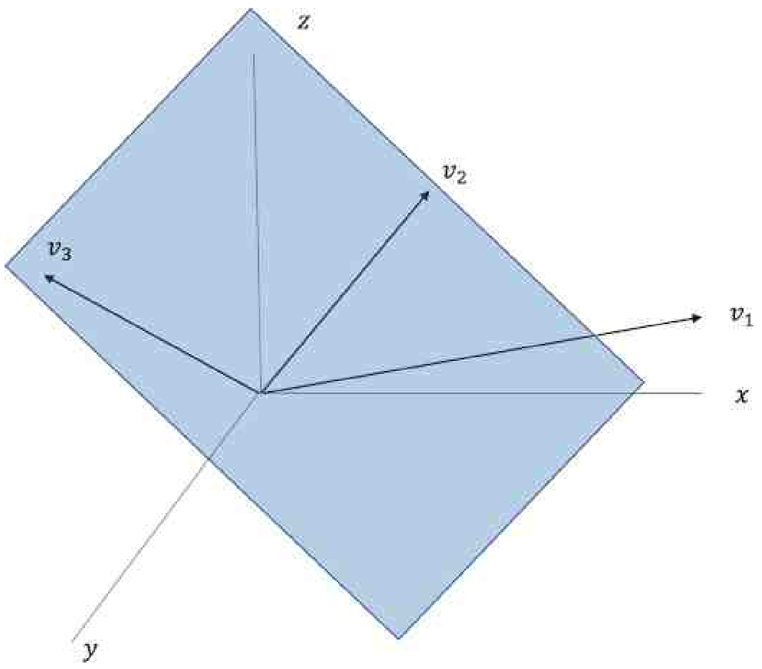


Figure 1-2: Three-dimensional vector space with vectors and a vector plane

Vector

An array of numbers, either continuous or discrete, is called a vector, and the space consisting of vectors is called a vector space. Vector space dimensions can be finite or infinite, but most machine-learning or data-science problems deal with fixed-length vectors; for example, the velocity of a car moving in the plane with velocities *Vx* and *Vy* in the *x* and *y* direction respectively (see Figure 1-3).

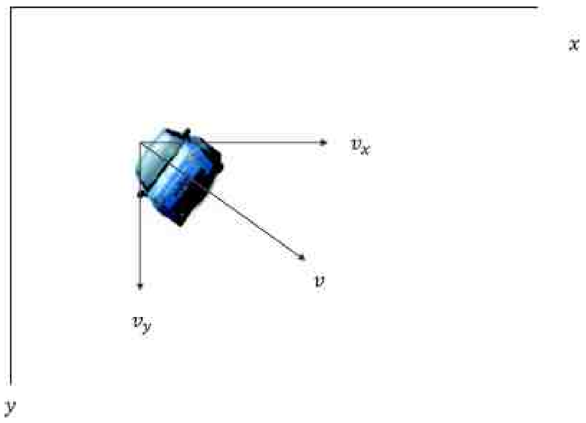


Figure 1-3: Car moving in the x-y vector plane with velocity components Vx and Vy

In machine learning, we deal with multidimensional data, so vectors become very crucial. Let's say we are trying to predict the housing prices in a region based on the area of the house, number of bedrooms, number of bathrooms, and population density of the locality. All these features form an input-feature vector for the housing price prediction problem.

Scalar

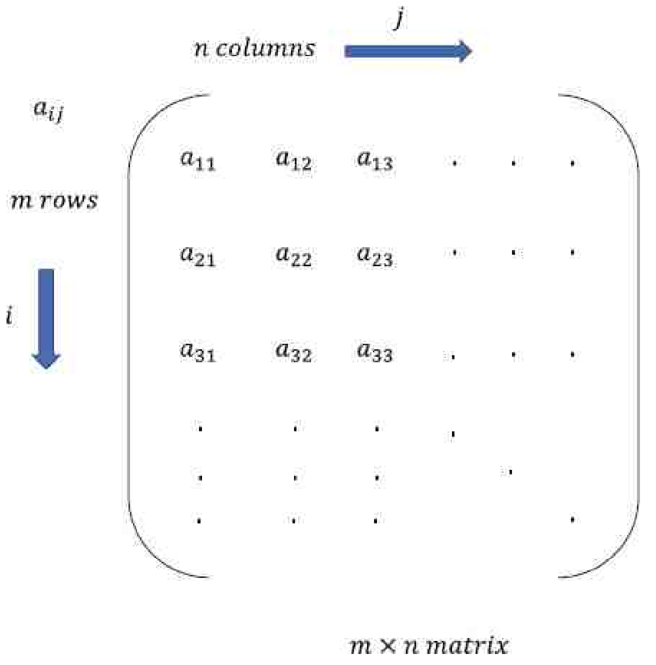
A one-dimensional vector is a scalar. As learned in high school, a scalar is a quantity that has only magnitude and no direction. This is because, since it has only one direction along which it can move, its direction is immaterial, and we are only concerned about the magnitude.

Examples: height of a child, weight of fruit, etc.

Matrix

A matrix is a two-dimensional array of numbers arranged in rows and columns. The size of the matrix is determined by its row length and column length. If a matrix *A* has *m* rows and *n* columns, it can be represented as a rectangular object (see Figure 1-

4a) having $m \times n$ elements, and it can be denoted as $A_{m \times n}$.



n columns
j
a_{ij}
m rows
i
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
$m \times n$ matrix

Figure 1-4a: Structure of a matrix

A few vectors belonging to the same vector space form a matrix.

For example, an image in grayscale is stored in a matrix form. The size of the image determines the image matrix size, and each matrix cell holds a value from 0-255 representing the pixel intensity. Illustrated in Figure 1-4b is a grayscale image followed by its matrix representation.

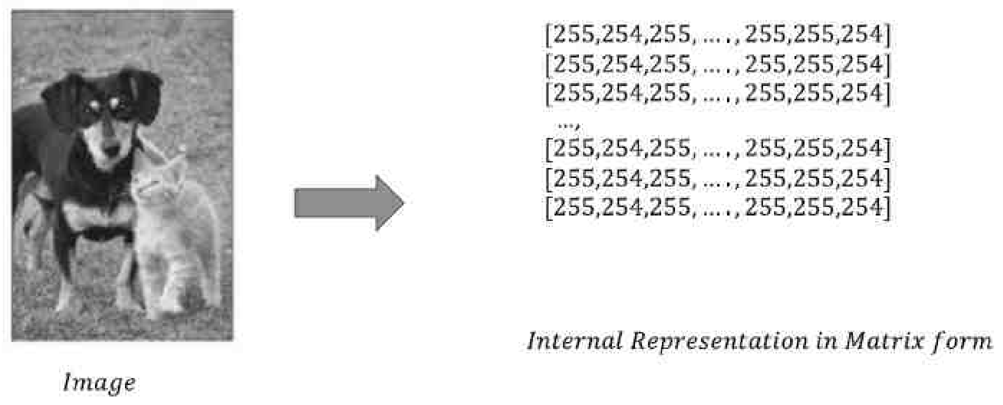


Figure 1-4b: Structure of a matrix

Tensor

A tensor is a multidimensional array of numbers. In fact, vectors and matrices can be treated as 1-D and 2-D tensors. In deep learning, tensors are mostly used for storing and processing data. For example, an image in RGB is stored in a three-dimensional tensor, where along one dimension we have the horizontal axis and along the other dimension we have the vertical axis, and where the third dimension corresponds to the three color channels, namely Red, Green, and Blue. Another example is the four-dimensional tensors used in feeding images through mini-batches in a convolutional neural network. Along the first dimension we have the image number in the batch and along the second dimension we have the color channels, and the third and fourth dimensions correspond to pixel location in the horizontal and vertical directions.

Matrix Operations and Manipulations

Most deep-learning computational activities are done through basic matrix operations, such as multiplication, addition, subtraction, transposition, and so forth. Hence, it makes sense to review the basic matrix operations.

A matrix A of m rows and n columns can be considered a matrix that contains n number of column vectors of dimension m

stacked side-by-side. We represent the matrix as

$A_{m\times n} \in \mathbb{R}^{m\times n}$

Addition of Two Matrices

The addition of two matrices *A* and *B* implies their element-wise addition. We can only add two matrices, provided their dimensions match. If *C* is the sum of matrices *A* and *B*, then

$$c_{ij} = a_{ij} + b_{ij} \quad \forall i \in \{1,2,..m\}, \forall j \in \{1,2,..n\}$$

$$where\ a_{ij} \in A, b_{ij} \in B, c_{ij} \in C$$

Example: $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$ then $A + B = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$

Subtraction of Two Matrices

The subtraction of two matrices *A* and *B* implies their element-wise subtraction. We can only subtract two matrices provided their dimensions match.

If *C* is the matrix representing *A* – *B*, then

$$c_{ij} = a_{ij} - b_{ij} \quad \forall i \in \{1,2,..m\}, \forall j \in \{1,2,..n\}$$

$$where\ a_{ij} \in A, b_{ij} \in B, c_{ij} \in C$$

Example: $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$ then $A - B = \begin{bmatrix} 1-5 & 2-6 \\ 3-7 & 4-8 \end{bmatrix} = \begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}$

Product of Two Matrices

For two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times q}$ to be multipliable, *n* should be equal to *p*. The resulting matrix is $C \in \mathbb{R}^{m\times q}$. The elements of *C* can be expressed as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad \forall i \in \{1,2,..m\}, \forall j \in \{1,2,..q\}$$

For example, the matrix multiplication of the two matrices $A,B \in \mathbb{R}^{2\times 2}$ can be computed as seen here:

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

$$c_{11} = \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 5 \\ 7 \end{bmatrix} = 1\times 5 + 2\times 7 = 19 \quad c_{12} = \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 6 \\ 8 \end{bmatrix} = 1\times 6 + 2\times 8 = 22$$

$$c_{21} = \begin{bmatrix} 3 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 7 \end{bmatrix} = 3\times 5 + 4\times 7 = 43 \quad c_{22} = \begin{bmatrix} 3 & 4 \end{bmatrix} \begin{bmatrix} 6 \\ 8 \end{bmatrix} = 3\times 6 + 4\times 8 = 50$$

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Transpose of a Matrix

The transpose of a matrix $A \in \mathbb{R}^{m\times n}$ is generally represented by $A^{T} \in \mathbb{R}^{n\times m}$ and is obtained by transposing the column vectors as row vectors.

$$a'_{ji} = a_{ij} \quad \forall\, i \in \{1,2,..m\}, \forall\, j \in \{1,2,..n\}$$

$$where\ a'_{ji} \in A^T\ and\ a_{ij} \in A$$

Example: $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ then $A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

The transpose of the product of two matrices *A* and *B* is the product of the transposes of matrices *A* and *B* in the reverse order; i.e., $(AB)^T = B^T A^T$

For example, if we take two matrices $A = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$ and $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, then

$(AB) = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 95 & 132 \\ 301 & 400 \end{bmatrix}$ and hence $(AB)^T = \begin{bmatrix} 95 & 301 \\ 132 & 400 \end{bmatrix}$

Now, $A^T = \begin{bmatrix} 19 & 43 \\ 22 & 50 \end{bmatrix}$ and $B^T = \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix}$

$$B^T A^T = \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix}\begin{bmatrix} 19 & 43 \\ 22 & 50 \end{bmatrix} = \begin{bmatrix} 95 & 301 \\ 132 & 400 \end{bmatrix}$$

Hence, the equality $(AB)^T = B^T A^T$ holds.

Dot Product of Two Vectors

Any vector of dimension *n* can be represented as a matrix $v \in \mathbb{R}^{n\times 1}$. Let us denote two *n* dimensional vectors $v_1 \in \mathbb{R}^{n\times 1}$ and $v_2 \in \mathbb{R}^{n\times 1}$.

$$v_1 = \begin{bmatrix} v_{11} \\ v_{12} \\ . \\ . \\ . \\ v_{1n} \end{bmatrix} v_2 = \begin{bmatrix} v_{21} \\ v_{22} \\ . \\ . \\ . \\ v_{2n} \end{bmatrix}$$

The dot product of two vectors is the sum of the product of corresponding components—i.e., components along the same dimension—and can be expressed as

$$v_1.v_2 = v_1{}^T v_2 = v_2{}^T v_1 = v_{11}v_{21} + v_{12}v_{22} + ..+ v_{1n}v_{2n} = \sum_{k=1}^{n} v_{1k}v_{2k}$$

Example: $v_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} v_2 = \begin{bmatrix} 3 \\ 5 \\ -1 \end{bmatrix}$ $v_1.v_2 = v_1{}^T v_2 = 1\times 3 + 2\times 5 - 3\times 1 = 10$

Matrix Working on a Vector

When a matrix is multiplied by a vector, the result is another vector. Let's say $A \in \mathbb{R}^{m\times n}$ is multiplied by the vector $x \in \mathbb{R}^{m\times 1}$. The result would produce a vector $b \in \mathbb{R}^{m\times 1}$

$$A = \begin{bmatrix} c_1^{(1)} c_1^{(2)} \ldots c_1^{(n)} \\ c_2^{(1)} c_2^{(2)} \ldots c_2^{(n)} \\ . \\ . \\ . \\ c_m^{(1)} c_m^{(2)} \ldots c_m^{(n)} \end{bmatrix} x = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

A consists of *n* column vectors $c^{(i)} \in \mathbb{R}^{m\times 1} \quad \forall i \in \{1,2,3,\ldots,n\}$.

$$A = \left[c^{(1)}c^{(2)}c^{(3)}\ldots.c^{(n)}\right]$$

$$b = Ax = \left[c^{(1)}c^{(2)}c^{(3)}\ldots.c^{(n)}\right]\begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{bmatrix} = x_1c^{(1)} + x_2c^{(2)} + \ldots. + x_nc^{(n)}$$

As we can see, the product is nothing but the linear combination of the column vectors of matrix *A*, with the components of vector *x* being the linear coefficients.

The new vector *b* formed through the multiplication has the same dimension as that of the column vectors of *A* and stays in the same column space. This is such a beautiful fact; no matter how we combine the column vectors, we can never leave the space spanned by the column vectors.

Now, let's work on an example.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad x = \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix} \qquad b = Ax = 2\begin{bmatrix} 1 \\ 4 \end{bmatrix} + 2\begin{bmatrix} 2 \\ 5 \end{bmatrix} + 3\begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 15 \\ 36 \end{bmatrix}$$

As we can see, both the column vectors of *A* and $b \in \mathbb{R}^{2\times 1}$

Linear Independence of Vectors

A vector is said to be linearly dependent on other vectors if it can be expressed as the linear combination of other vectors.

If $v_1 = 5v_2 + 7v_3$, then v_1, v_2 and v_3 are not linearly independent since at least one of them can be expressed as the sum of other vectors. In general, a set of *n* vectors $v_1, v_2, v_3\ldots, v_n \in \mathbb{R}^{m\times 1}$ is said to be linearly independent if and only if $a_lv_l + a_2v_2 + a_3v_3 + \ldots + a_nv_n = 0$ implies each of $a_i = 0$ $\forall i \in \{1,2,\ldots n\}$.

If $a_lv_l + a_2v_2 + a_3v_3 + \ldots + a_nv_n = 0$ and not all $a_i = 0$, then the vectors are not linearly independent.

Given a set of vectors, the following method can be used to check whether they are linearly independent or not.

$a_lv_l + a_2v_2 + a_3v_3 + \ldots + a_nv_n = 0$ can be written as

$$\left[v_1v_2\ldots.v_n\right]\begin{bmatrix} a_1 \\ a_2 \\ . \\ . \\ a_n \end{bmatrix} = 0 \, where \; v_i \in \mathbb{R}^{m\times 1} \, \forall i \in \{1,2,\ldots,n\}, \begin{bmatrix} a_1 \\ a_2 \\ . \\ . \\ a_n \end{bmatrix} \in \mathbb{R}^{n\times 1}$$

Solving for $[a_1 \; a_2 \; \ldots. \; a_n]^T$, if the only solution we get is the zero vector, then the set of vectors $v_1, v_2, \ldots. v_n$ is said to be linearly independent.

If a set of *n* vectors $v_i \in \mathbb{R}^{n\times 1}$ is linearly independent, then those vectors span the whole *n*-dimensional space. In other words, by taking linear combinations of the *n* vectors, one can produce all possible vectors in the *n*-dimensional space. If the *n* vectors are not linearly independent, they span only a subspace within the *n*-dimensional space.

To illustrate this fact, let us take vectors in three-dimensional space, as illustrated in Figure 1-5.

If we have a vector $v_1 = [123]^T$, we can span only one dimension in the three-dimensional space because all the vectors that can be formed with this vector would have the same direction as that of v_1, with the magnitude being determined by the scaler multiplier. In other words, each vector would be of the form a_1v_1.

Now, let's take another vector v_2 =[597]^T, whose direction is not the same as that of v_1. So, the span of the two vectors *Span*(v_1, v_2) is nothing but the linear combination of v_1 and v_2. With these two vectors, we can form any vector of the form av_l + bv_2 that lies in the plane of the two vectors. Basically, we will span a two-dimensional subspace within the three-dimensional space. The same is illustrated in the following diagram.

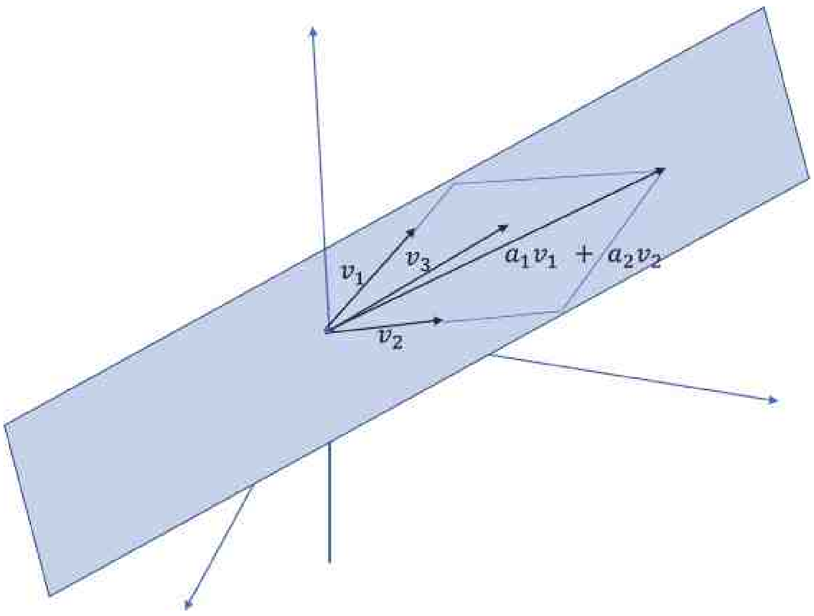


Figure 1-5: A two-dimensional subspace spanned by v_1 and v_2 in a three-dimensional vector space

Let's us add another vector v_3 = [481]^T to our vector set. Now, if we consider the *Span*(v_1, v_2 v_3), we can form any vector in the three-dimensional plane. You take any three-dimensional vector you wish, and it can be expressed as a linear combination of the preceding three vectors.

These three vectors form a basis for the three-dimensional space. Any three linearly independent vectors would form a basis for the three-dimensional space. The same can be generalized for any *n*-dimensional space.

If we had taken a vector v_3, which is a linear combination of v_1 and v_2, then it wouldn't have been possible to span the whole three-dimensional space. We would have been confined to the two-dimensional subspace spanned by v_1 and v_2.

Rank of a Matrix

One of the most important concepts in linear algebra is the rank of a matrix. The rank of a matrix is the number of linearly independent column vectors or row vectors. The number of independent columns vectors would always be equal to the number of independent row vectors for a matrix.

Example - Consider the matrix $A=\begin{bmatrix}1 & 3 & 4\\ 2 & 5 & 7\\ 3 & 7 & 10\end{bmatrix}$

The column vectors $\begin{bmatrix}1\\2\\3\end{bmatrix}$ and $\begin{bmatrix}3\\5\\7\end{bmatrix}$ are linearly independent. However, $\begin{bmatrix}4\\7\\10\end{bmatrix}$ is not linearly independent since it's the linear combination of the other two column vectors; i.e., $\begin{bmatrix}4\\7\\10\end{bmatrix}=\begin{bmatrix}1\\2\\3\end{bmatrix}+\begin{bmatrix}3\\5\\7\end{bmatrix}$. Hence, the rank of the matrix is 2 since it has two linearly independent column vectors.

As the rank of the matrix is 2, the column vectors of the matrix can span only a two-dimensional subspace inside the three-dimensional vector space. The two-dimensional subspace is the one that can be formed by taking the linear combination of $\begin{bmatrix}1\\2\\3\end{bmatrix}$

and $\begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix}$.

A few important notes:

- A square matrix $A \in \mathbb{R}^{n \times n}$ is said to be full rank if the rank of A is n. A square matrix of rank n implies that all the n column vectors and even the n row vectors for that matter are linearly independent, and hence it would be possible to span the whole n-dimensional space by taking the linear combination of the n column vectors of the matrix A.

- If a square matrix $A \in \mathbb{R}^{n \times n}$ is not full rank, then it is a singular matrix; i.e., all its column vectors or row vectors are not linearly independent. A singular matrix has an undefined matrix inverse and zero determinant.

Identity Matrix or Operator

A matrix $I \in \mathbb{R}^{n \times n}$ is said to be an identity matrix or operator if any vector or matrix when multiplied by I remains unchanged. A 3×3 identity matrix is given by

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \in \mathbb{R}^{3 \times 3}$$

Let's say we take the vector $v = [2\ 3\ 4]^T$

$$Iv = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

Similarly, let's say we have a matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

The matrices AI and IA are both equal to matrix A. Hence, the matrix multiplication is commutative when one of the matrices is an identity matrix.

Determinant of a Matrix

A determinant of a square matrix A is a number and is denoted by $det(A)$. It can be interpreted in several ways. For a matrix $A \in \mathbb{R}^{n \times n}$ the determinant denotes the n-dimensional volume enclosed by the n row vectors of the matrix. For the determinant to be non-zero, all the column vectors or the row vectors of A should be linearly independent. If the n row vectors or column vectors are not linearly independent, then they don't span the whole n-dimensional space, but rather a subspace of dimension less than n, and hence the n-dimensional volume is zero. For a matrix $A \in \mathbb{R}^{2 \times 2}$ the determinant is expressed as

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \in \mathbb{R}^{2 \times 2}$$

$$det(A) = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

Similarly, for a matrix $B \in \mathbb{R}^{3 \times 3}$ the determinant of the matrix is given by

$$B=\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \in \mathbb{R}^{3\times3}$$

$$det(B)=a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

$$\text{where } det\left(\begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix}\right)=\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix}$$

The method for determinant computation can be generalized to $\mathbb{R}^{n\times n}$ matrices. Treating *B* as an *n*-dimensional matrix, its determinant can be expressed as

$$det(B)=\begin{bmatrix} a_{11}\times & & \\ & a_{22} & a_{23} \\ & a_{32} & a_{33} \end{bmatrix} - \begin{bmatrix} & a_{12}\times & \\ a_{21} & & a_{23} \\ a_{31} & & a_{33} \end{bmatrix} + \begin{bmatrix} & & a_{13}\times \\ a_{31} & a_{22} & \\ a_{31} & a_{32} & \end{bmatrix}$$

For example, the determinant of the matrix $A=\begin{bmatrix} 6 & 1 & 1 \\ 4 & -2 & 5 \\ 2 & 8 & 7 \end{bmatrix}$ can be computed as follows:

$$det(A)=\begin{bmatrix} 6\times & & \\ & -2 & 5 \\ & 8 & 7 \end{bmatrix} - \begin{bmatrix} & 1\times & \\ 4 & & 5 \\ 2 & & 7 \end{bmatrix} + \begin{bmatrix} & & 1\times \\ 4 & -2 & \\ 2 & 8 & \end{bmatrix}$$

$$=6\times\begin{vmatrix} -2 & 5 \\ 8 & 7 \end{vmatrix} - 1\times\begin{vmatrix} 4 & 5 \\ 2 & 7 \end{vmatrix} + 1\times\begin{vmatrix} 4 & -2 \\ 2 & 8 \end{vmatrix} = 6(-14-40)-1(28-10)+1(32+4)$$

$$=6\times(-54)-1(18)+36=-306$$

Interpretation of Determinant

As stated earlier, the absolute value of the determinant of a matrix determines the volume enclosed by the row vectors acting as edges.

For a matrix $A \in \mathbb{R}^{2\times2}$, it denotes the area of the parallelogram with the two-row vector acting as edges.

For a matrix $A=\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, the $det(A)$ is equal to the area of the parallelogram with vectors $u = [a\ b]^T$ and $v = [c\ d]^T$ as edges.

Area of the parallelogram = $|u|\,|v|\sin\theta$ where θ is the angle between *u* and *v* (see Figure 1-6).

$$=\sqrt{a^2+b^2}\sqrt{c^2+d^2}\frac{(ad-bc)}{\sqrt{a^2+b^2}\sqrt{c^2+d^2}}=(ad-bc)$$

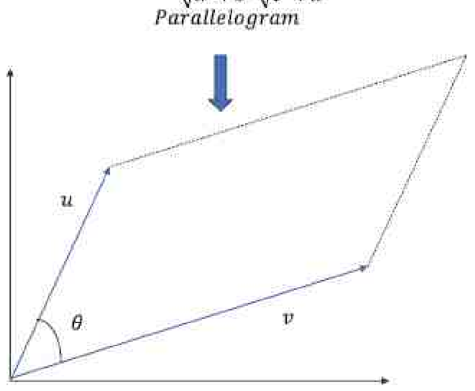


Figure 1-6: Parallelogram formed by two vectors

Similarly, for a matrix $B \in \mathbb{R}^{3\times3}$, the determinant is the volume of the parallelepiped with the three-row vectors as edges.

Inverse of a Matrix

An inverse of a square matrix $A \in \mathbb{R}^{n \times n}$ is denoted by A^{-1} and produces the identity matrix $I \in \mathbb{R}^{n \times n}$ when multiplied by A.

$$AA^{-1} = A^{-1}A = I$$

Not all square matrices have inverses for A. The formula for computing the inverse of A is as follows:

$$A^{-1} = \frac{adjoint(A)}{\det(A)} = \frac{(cofactor\ matrix\ of\ A)^T}{\det(A)}$$

If a square matrix $A \in \mathbb{R}^{n \times n}$ is singular—i.e., if A doesn't have n independent column or row vectors—then the inverse of A doesn't exist. This is because for a singular matrix $det(A) = 0$ and hence the inverse becomes undefined.

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Let the elements of A be represented by a_{ij}, where i represents the row number and j the column number for an element.

Then, the cofactor for $a_{ij} = (-1)^{i+j} d_{ij}$, where d_{ij} is the determinant of the matrix formed by deleting the row i and the column j from A.

The cofactor for the element $a = (-1)^{1+1} \begin{vmatrix} e & f \\ h & i \end{vmatrix} = ei - fh$.

Similarly, the cofactor for element $b = (-1)^{1+2} \begin{vmatrix} d & f \\ g & i \end{vmatrix} = -(di - fg)$.

Once the cofactor matrix is formed, the transpose of the cofactor matrix would give us *adjoint*(A). The *adjoint*(A) divided by the *det*(A) gives A^{-1}.

For example, the inverse matrix of $A = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}$ can be computed as follows:

Cofactor matrix of $A = \begin{bmatrix} 1(2) & -1(3) \\ -1(3) & 1(4) \end{bmatrix} = \begin{bmatrix} 2 & -3 \\ -3 & 4 \end{bmatrix}$

$$det(A) = \begin{vmatrix} 4 & 3 \\ 3 & 2 \end{vmatrix} = 8 - 9 = -1 \text{ Therefore, } A^{-1} = \frac{(cofactor\ matrix\ of\ A)^T}{\det(A)} = \frac{\begin{bmatrix} 2 & -3 \\ -3 & 4 \end{bmatrix}^T}{-1} = \begin{bmatrix} -2 & 3 \\ 3 & -4 \end{bmatrix}.$$

A few rules for inverses of a matrix:

- $(AB)^{-1} = B^{-1}A^{-1}$
- $I^{-1} = I$, where I is the identity matrix

Norm of a Vector

The norm of a vector is a measure of its magnitude. There are several kinds of such norms. The most familiar is the Euclidean norm, defined next. It is also known as the l^2 norm.

For a vector $x \in \mathbb{R}^{n \times 1}$ the l^2 norm is as follows:

$$\|x\|_2 = \left(|x_1|^2 + |x_2|^2 + \ldots + |x_n|^2\right)^{1/2} = (x.x)^{1/2} = (x^T x)^{1/2}$$

Similarly, the l^1 norm is the sum of the absolute values of the vector components.

$$\|x\|_1 = |x_1| + |x_2| + \ldots + |x_n|$$

In general, the l^p norm of a vector can be defined as follows when $1 < p < \infty$:

$$\left(|x_1|^p + |x_2|^p + \ldots + |x_n|^p\right)^{1/p}$$

When $p \to \infty$ then the norm is called Supremum norm and is defined as follows:

$$\lim_{p\to\infty} \|x\|_p = \lim_{p\to\infty}\left(|x_1|^p + |x_2|^p + \ldots + |x_n|^p\right)^{1/p}$$

$$= max\left(x_1, x_2, \ldots., x_n\right)$$

In Figure 1-7, the unit norm curves have been plotted for l^1, l^2 and Supremum norm.

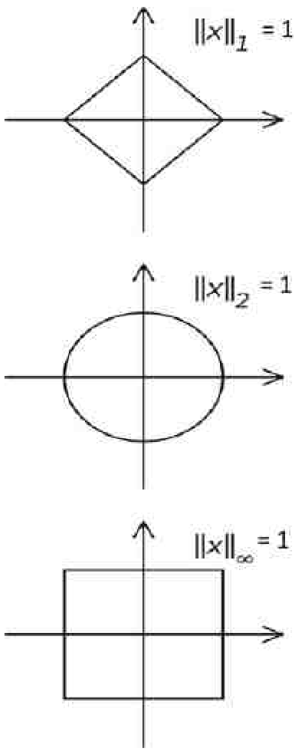


Figure 1-7: Unit l^1,l^2 and Supremum norms of vectors $\in \mathbb{R}^{2\times l}$

Generally, for machine learning we use both l^2 and l^1 norms for several purposes. For instance, the least square cost function that we use in linear regression is the l^2 norm of the error vector; i.e., the difference between the actual target-value vector and the predicted target-value vector. Similarly, very often we would have to use regularization for our model, with the result that the model doesn't fit the training data very well and fails to generalize to new data. To achieve regularization, we generally add the square of either the l^2 norm or the l^1 norm of the parameter vector for the model as a penalty in the cost function for the model. When the l^2 norm of the parameter vector is used for regularization, it is generally known as Ridge Regularization, whereas when the l^1 norm is used instead it is known as Lasso Regularization.

Pseudo Inverse of a Matrix

If we have a problem $Ax = b$ where $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^{n\times 1}$ are provided and we are required to solve for $x \in \mathbb{R}^{n\times 1}$, we can solve for x as $x = A^{-1}b$ provided A is not singular and its inverse exists.

However, if $A \in \mathbb{R}^{m\times n}$—i.e., if A is a rectangular matrix and $m > n$ —then A^{-1} doesn't exist, and hence we can't solve for x by the preceding approach. In such cases, we can get an optimal solution, as $x^* = (A^TA)^{-1} A^Tb$. The matrix $(A^TA)^{-1} A^T$ is called the pseudo-inverse since it acts as an inverse to provide the optimal solution. This pseudo-inverse would come up in least square techniques, such as linear regression.

Unit Vector in the Direction of a Specific Vector

Unit vector in the direction of the specific vector is the vector divided by its magnitude or norm. For a Euclidian space, also called an l^2 space, the unit vector in the direction of the vector $x = [3\ 4]^T$ is

$$\frac{x}{\|x\|_2} = \frac{x}{\left(x^T x\right)^{1/2}} = \frac{[3\,4]^T}{5} = [0.6\,0.8]^T$$

Projection of a Vector in the Direction of Another Vector

Projection of a vector v_1 in the direction of v_2 is the dot product of v_1 with the unit vector in the direction of v_2. $\|v_{12}\| = v_1^T u_2$, where $||v_{12}||$ is the projection of v_1 onto v_2 and u_2 is the unit vector in the direction of v_2. Since $u_2 = \frac{v_2}{\|v_2\|_2}$ as per the definition of a unit vector, the projection can also be expressed as $\|v_{12}\| = v_1^T u_2 = v_1^T \frac{v_2}{\|v_2\|_2} = v_1^T \frac{v_2}{\left(v_2^T v_2\right)^{1/2}}$

For example, the projection of the vector $[1\ 1]^T$ in the direction of vector $[3\ 4]^T$ is the dot product of $[1\ 1]^T$ with the unit vector in the direction of $[3\ 4]^T$; i.e., $[0.6\ 0.8]^T$ as computed earlier.

The required projection $= [1 1]^T \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix} = 1\times 0.6 + 1\times 0.8 = 1.4$.

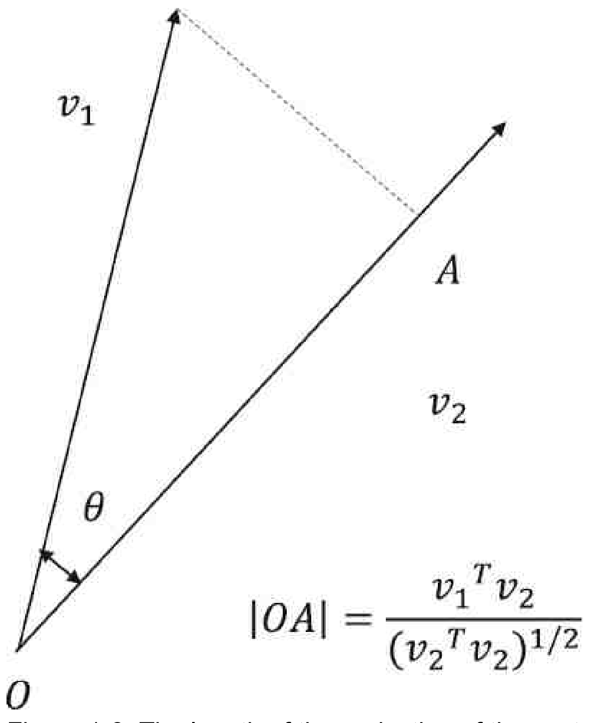


Figure 1-8: The length of the projection of the vector v_1 onto v_2

In Figure 1-8, the length of the line segment *OA* gives the length of the projection of the vector v_1 onto v_2.

Eigen Vectors

Here we come to one of the most important concepts in linear algebra—Eigen vectors and Eigen values. Eigen values and Eigen vectors come up in several areas of machine learning. For example, the principal components in principal-component analysis are the Eigen vectors of the covariance matrix, while the Eigen values are the covariances along the principal components. Similarly, in Google's page-rank algorithm the vector of the page-rank score is nothing but an Eigen vector of the page transition probability matrix corresponding to the Eigen value of 1.

A matrix works on a vector as an operator. The operation of the matrix on the vector is to transform the vector into another vector whose dimensions might or might not be same as the original vector based on the matrix dimension.

When a matrix $A \in \mathbb{R}^{n\times n}$ works on a vector $x \in \mathbb{R}^{n\times 1}$, we again get back a vector $Ax \in \mathbb{R}^{n\times 1}$. Generally, the magnitude as well as the direction of the new vector is different from that of the original vector. If in such a scenario the newly generated vector has the same direction or exactly the opposite direction as that of the original vector, then any vector in such a direction is called an Eigen vector. The magnitude by which the vector gets stretched is called the Eigen value (see Figure 1-9).

$$Ax = \lambda x$$

where *A* is the matrix operator operating on the vector *v* by multiplication, which is also the Eigen vector, and *λ* is the Eigen value.

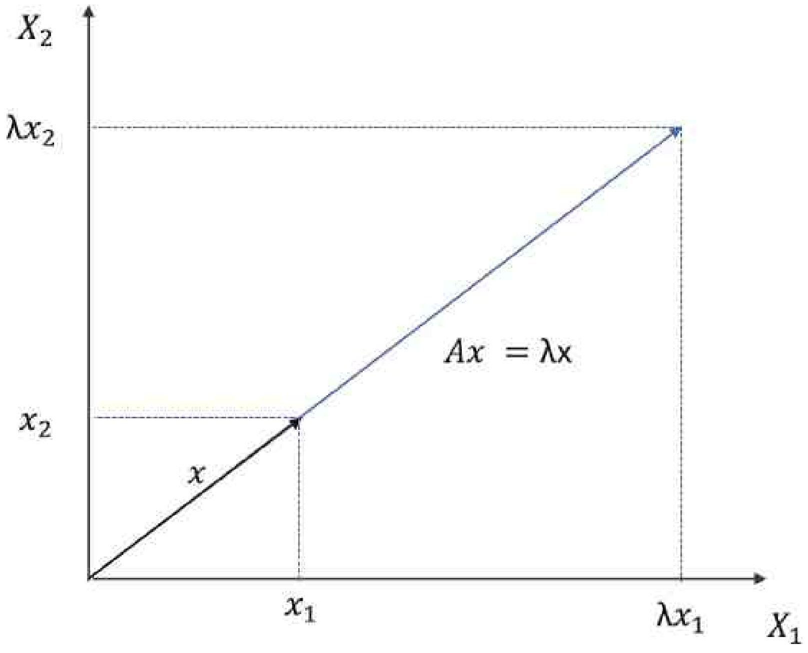


Figure 1-9: Eigen vector unaffected by the matrix transformation A

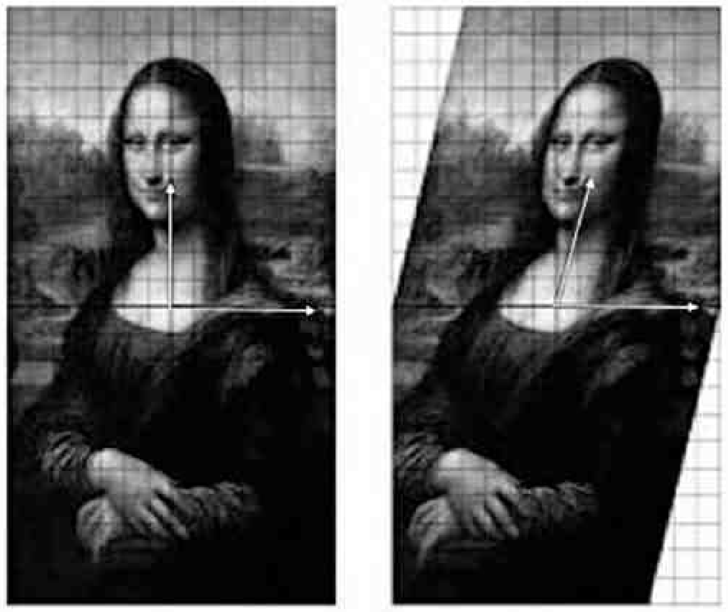


Figure 1-10: The famous Mona Lisa image has a transformation applied to the vector space of pixel location

As we can see from Figure 1-10, the pixels along the horizontal axis represented by a vector have changed direction when a transformation to the image space is applied, while the pixel vector along the horizontal direction hasn't changed direction. Hence, the pixel vector along the horizontal axis is an Eigen vector to the matrix transformation being applied to the *Mona Lisa* image.

Characteristic Equation of a Matrix

The roots of the characteristic equation of a matrix $A \in \mathbb{R}^{n \times n}$ gives us the Eigen values of the matrix. There would be *n* Eigen values corresponding to *n* Eigen vectors for a square matrix of order *n*.

For an Eigen vector $v \in \mathbb{R}^{n\times 1}$ corresponding to an Eigen value of λ, we have

$$Av = \lambda v$$

$$=> (A - \lambda I)v = 0$$

Now, *v* being an Eigen vector is non-zero, and hence (*A* – λ*I*) must be singular for the preceding to hold true.

For (*A* – λ*I*) to be singular, *det*(*A* – λ*I*) = 0, which is the characteristics equation for matrix *A*. The roots of the characteristics equation gives us the Eigen values. Substituting the Eigen values in the *Av* = λ*v* equation and then solving for *v* gives the Eigen vector corresponding to the Eigen value.

For example, the Eigen values and Eigen vectors of the matrix

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

can be computed as seen next.

The characteristics equation for the matrix *A* is det(*A* – λ*I*) = 0.

$$\begin{vmatrix} -\lambda & 1 \\ -2 & -3-\lambda \end{vmatrix} = 0 \quad => \lambda^2 + 3\lambda + 2 = 0 => \lambda = -2, -1$$

The two Eigen values are −2 and −1.

Let the Eigen vector corresponding to the Eigen value of −2 be *u* = [*a b*]^T.

$$\begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix} = -2\begin{bmatrix} a \\ b \end{bmatrix}$$

This gives us the following two equations:

$$0a + 1b = -2a \quad => 2a + b = 0 \qquad - \; (1)$$

$$-2a - 3b = -2b => 2a + b = 0 \qquad - \; (2)$$

Both the equations are the same; i.e., $2a + b = 0 => \frac{a}{b} = \frac{1}{-2}$.

Let *a* = k_1 and *b* = −2k_l, where k_1 is a constant.

Therefore, the Eigen vector corresponding to the Eigen value -2 is $u = k_1\begin{bmatrix} 1 \\ -2 \end{bmatrix}$.

Using the same process, the Eigen vector *v* corresponding to the Eigen value of −1 is $v = k_2\begin{bmatrix} 1 \\ -1 \end{bmatrix}$.

One thing to note is that Eigen vectors and Eigen values are always related to a specific operator (in the preceding case, matrix *A* is the operator) working on a vector space. Eigen values and Eigen vectors are not specific to any vector space.

Functions can be treated as vectors. Let's say we have a function *f*(*x*) = e^{ax}.

Each of the infinite values of *x* would be a dimension, and the value of *f*(*x*) evaluated at those values would be the vector component along that dimension. So, what we would get is an infinite vector space.

Now, let's look at the differentiator operator.

$$\frac{dy}{dx}(f(x)) = \frac{dy}{dx}(e^{ax}) = ae^{ax}$$

Here, $\frac{dy}{dx}$ is the operator and e^{ax} is an Eigen function with respect to the operator, while *a* is the corresponding Eigen value.

As expressed earlier, the applications of Eigen vectors and Eigen values are profound and far reaching in almost any domain, and this is true for machine learning as well. To get an idea of how Eigen vectors have influenced modern applications, we will look at the Google page-ranking algorithm in a simplistic setting.

Let us look at the page-ranking algorithm for a simple website that has three pages—A, B, and C—as illustrated in Figure 1-11.

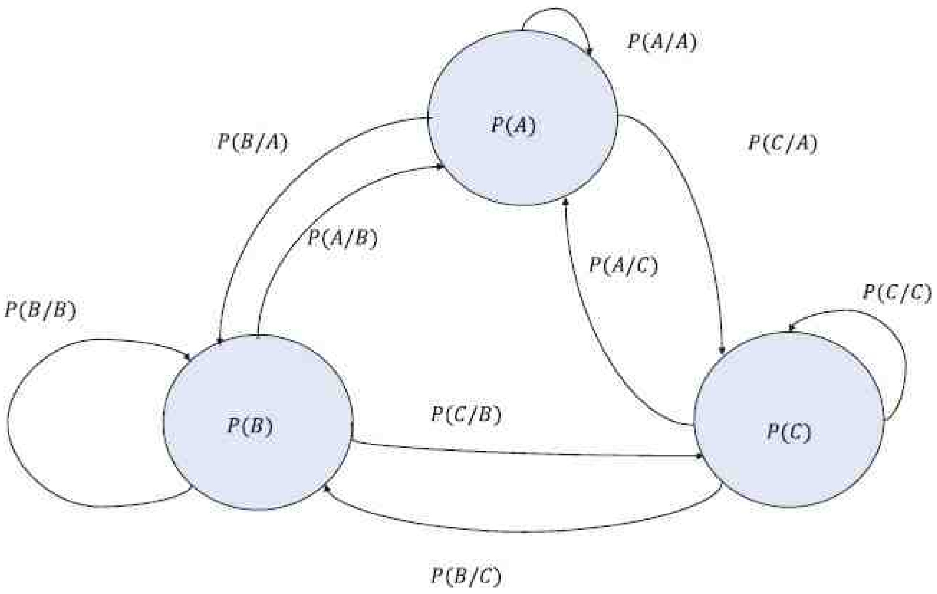


Figure 1-11: Transition probability diagram for three pages A, B, and C

In a web setting, one can jump from one page to another page given that the original page has a link to the next page. Also, a page can self-reference and have a link to itself. So, if a user goes from page A to B because page A references page B, the event can be denoted by *B/A*. *P*(*B/A*) can be computed by the total number of visits to page B from page A divided by the total number of visits to page A. The transition probabilities for all page combinations can be computed similarly. Since the probabilities are computed by normalizing count, the individual probabilities for pages would carry the essence of the importance of the pages.

In the steady state, the probabilities of each page would become constant. We need to compute the steady-state probability of each page based on the transition probabilities.

For the probability of any page to remain constant at steady state, probability mass going out should be equal to probability mass coming in, and each of them—when summed up with probability mass that stays in a page—should equal the probability of the page. In that light, if we consider the equilibrium equation around page A, the probability mass going out of *A* is *P*(*B / A*)*P*(*A*) + *P*(*C / A*)*P* (*A*) whereas the probability mass coming into A is *P*(*A / B*)*P*(*B*) + *P*(*A / C*)*P*(*C*). The probability mass *P*(*A/A*)*P*(*A*) remains at A itself. Hence, at equilibrium the sum of probability mass coming from outside—i.e., *P* (*A / B*)*P*(*B*) + *P* (*A / C*)*P* (*C*)—and probability mass remaining at A—i.e., *P*(*A/A*)*P*(*A*)—should equal *P*(*A*), as expressed here:

(1) $P(A/A)P(A)+P(A/B)P(B)+P(A/C)P(C)=P(A)$

Similarly, if we consider the equilibrium around pages B and C, the following holds true:

(2) $P(B/A)P(A)+P(B/B)P(B)+P(B/C)P(C)=P(B)$
(3) $P(C/A)P(A)+P(C/B)P(B)+P(C/C)P(C)=P(C)$

Now comes the linear algebra part. We can arrange the three equations into a matrix working on a vector, as follows:

$$\begin{bmatrix} P(A/A) & P(A/B) & P(A/C) \\ P(B/A) & P(B/B) & P(B/C) \\ P(C/A) & P(C/B) & P(C/C) \end{bmatrix} \begin{bmatrix} P(A) \\ P(B) \\ P(C) \end{bmatrix} = \begin{bmatrix} P(A) \\ P(B) \\ P(C) \end{bmatrix}$$

The transition-probability matrix works on the page-probability vector to produce again the page-probability vector. The page-probability vector, as we can see, is nothing but an Eigen vector to the page-transition-probability matrix, and the corresponding Eigen value for the same is 1.

So, computing the Eigen vector corresponding to the Eigen value of 1 would give us the page-probability vector, which in turn can be used to rank the pages. Several page-ranking algorithms of reputed search engines work on the same principle. Of course, the actual algorithms of the search engines have several modifications to this naïve model, but the underlying concept is the same. The probability vector can be determined through methods such as power iteration, as discussed in the next section.

Power Iteration Method for Computing Eigen Vector

The power iteration method is an iteration technique used to compute the Eigen vector of a matrix corresponding to the Eigen value of largest magnitude.

Let $A \in \mathbb{R}^{n \times n}$ and then let that the n Eigen values in order of magnitude are $\lambda_1 > \lambda_2 > \lambda_3 > \ldots > \lambda_n$ and the corresponding Eigen vectors are $v_1 > v_2 > v_3 > \ldots > v_n$.

Power iteration starts with a random vector v, which should have some component in the direction of the Eigen vector corresponding to the largest Eigen value; i.e., v_1.

The approximate Eigen vector in any iteration is given by

$$v^{(k+1)} = \frac{Av^{(k)}}{\left\|Av^{(k)}\right\|}$$

After a sufficient number of iterations, $v^{(k+1)}$ converges to v_1. In every iteration, we multiply the matrix A by the vector obtained from the prior step. If we remove the normalizing of the vector to convert it to a unit vector in the iterative method, we have $v^{(k+1)} = A^k v$.

Let the initial vector v be represented as a combination of the Eigen vectors: $v = k_l v_l + k_2 v_2 + .. + k_n v_n$ where $k_i \ \forall \ i \in \{l,2,3,..n\}$ are constants.

$$\begin{aligned} v^{(k+1)} &= A^k v = A^k \left(k_1 v_1 + k_2 v_2 + .. + k_n v_n\right) \\ &= k_1 A^k v_1 + k_2 A^k v_2 + .. + k_n A^k v_n \\ &= k_1 \lambda_1^{\ k} v_1 + k_2 \lambda_2^{\ k} v_2 + .. + k_n \lambda_n^{\ k} v_n \\ &= \lambda_1^{\ k} \left(k_1 v_1 + k_2 \left(\frac{\lambda_2}{\lambda_1}\right)^k v_2 + .. + k_n \left(\frac{\lambda_n}{\lambda_1}\right)^k v_n \right) \end{aligned}$$

Now, when k is sufficiently large—i.e., $(k \to \infty)$—all the terms except the first will vanish since

$$\left(\frac{\lambda_i}{\lambda_1}\right)^k \to 0 \ \forall \, i \in \{2,3,..n\}$$

Therefore, $v^{(k+1)} = \lambda_1^{\ k} k_1 v_1$, which gives us the Eigen vector corresponding to the Eigen value of largest magnitude. The rate of convergence depends on the magnitude of the second-largest Eigen value in comparison to the largest Eigen value. The method converges slowly if the second-largest Eigen value is close in magnitude to the largest one.

Note In this chapter, I have touched upon the basics of linear algebra so that readers who are not familiar with this subject have some starting point. However, i would suggest the reader to take up linear algebra in more detail in his or her spare time. Renowned professor Gilbert strang's book *Linear Algebra and Its Applications* is a wonderful way to get started.

Calculus

In its very simplest form, calculus is a branch of mathematics that deals with differentials and integrals of functions. Having a good understanding of calculus is important for machine learning for several reasons:

- Different machine-learning models are expressed as functions of several variables.
- To build a machine-learning model, we generally compute a cost function for the model based on the data and model parameters, and through optimization of the cost function we derive the model parameters that best explain the given

data.

Differentiation

Differentiation of a function generally means the rate of change of a quantity represented by a function with respect to another quantity on which the function is dependent on.

Let's say a particle moves in a one-dimensional plane—i.e., a straight line—and its distance at any specific time is defined by the function $f(t) = 5t^2$.

The velocity of the particle at any specific time would be given by the derivative of the function with respect to time *t*.

The derivative of the function is defined as $\frac{df(t)}{dt}$ and is generally expressed by the following formulae based on whichever is convenient:

$$\frac{df}{dt} = \lim_{h \to 0} \frac{f(t+h) - f(t)}{h}$$

or

$$\frac{df}{dt} = \lim_{h \to 0} \frac{f(t+h) - f(t-h)}{2h}$$

When we deal with a function that is dependent on multiple variables, the derivative of the function with respect to each of the variables keeping the others fixed is called a partial derivative, and the vector of partial derivatives is called the gradient of the function.

Let's say the price *z* of a house is dependent on two variables: square feet area of the house *x* and the number of bedrooms *y*.

$$z = f(x,y)$$

The partial derivative of *z* with respect to *x* is represented by

$$\frac{\partial z}{\partial x} = \lim_{h \to 0} \frac{f(x+h,y) - f(x,y)}{h}$$

Similarly, the partial derivative of *z* with respect to *y* is

$$\frac{\partial z}{\partial y} = \lim_{h \to 0} \frac{f(x,y+h) - f(x,y)}{h}$$

Bear in mind that in a partial derivate, except the variable with respect to which the derivate is being taken, are held constant.

Gradient of a Function

For a function with two variables $z = f(x,y)$, the vector of partial derivatives $\begin{bmatrix} \frac{\partial z}{\partial x} & \frac{\partial z}{\partial y} \end{bmatrix}^T$ is called the gradient of the function and is denoted by ∇z. The same can be generalized for a function with *n* variables. A multivariate function $f(x_1, x_2, .., x_n)$ can also be expressed as $f(x)$, where $x = [x_1, x_2 \ldots x_n]^T \in \mathbb{R}^{n \times 1}$. The gradient vector for the multivariate function $f(x)$ with respect to *x* can be expressed as $\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \frac{\partial f}{\partial x_2} \ldots\ldots \frac{\partial f}{\partial x_n} \end{bmatrix}^T$.

For example, the gradient of a function with three variables $f(x, y, z) = x + y^2 + z^3$ is given by

$$\nabla f = \begin{bmatrix} 1 \, 2y \, 3z^2 \end{bmatrix}^T$$

The gradient and the partial derivatives are important in machine-learning algorithms when we try to maximize or minimize cost functions with respect to the model parameters, since at the maxima and minima the gradient vector of a function is zero. At the maxima and minima of a function, the gradient vector of the function should be a zero vector.

Successive Partial Derivatives

We can have successive partial derivatives of a function with respect to multiple variables. For example, for a function *z* = *f*(*x,y*)

$$\frac{\partial}{\partial y}\left(\frac{\partial z}{\partial x}\right)=\frac{\partial^2 z}{\partial y \partial x}$$

This is the partial derivative of *z* with respect to *x* first and then with respect to *y*. Similarly,

$$\frac{\partial}{\partial x}\left(\frac{\partial z}{\partial y}\right)=\frac{\partial^2 z}{\partial x \partial y}$$

If the second derivatives are continuous, the order of partial derivatives doesn't matter and

$$\frac{\partial^2 z}{\partial x \partial y}=\frac{\partial^2 z}{\partial y \partial x}.$$

Hessian Matrix of a Function

The Hessian of a multivariate function is a matrix of second-order partial derivatives. For a function *f*(*x, y, z*), the Hessian is defined as follows:

$$Hf=\begin{bmatrix} \frac{\delta^2 f}{\delta x^2} & \frac{\delta^2 f}{\delta x \delta y} & \frac{\delta^2 f}{\delta x \delta z} \\ \frac{\delta^2 f}{\delta y \delta x} & \frac{\delta^2 f}{\delta y^2} & \frac{\delta^2 f}{\delta y \delta z} \\ \frac{\delta^2 f}{\delta z \delta x} & \frac{\delta^2 f}{\delta z \delta y} & \frac{\delta^2 f}{\delta z^2} \end{bmatrix}$$

The Hessian is useful in the optimization problems that we come across so frequently in the machine-learning domain. For instance, in minimizing a cost function to arrive at a set of model parameters, the Hessian is used to get better estimates for the next set of parameter values, especially if the cost function is non-linear in nature. Non-linear optimization techniques, such as Newton's method, Broyden-Fletcher-Goldfarb-Shanno (BFGS), and its variants, use the Hessian for minimizing cost functions.

Maxima and Minima of Functions

Evaluating the maxima and minima of functions has tremendous applications in machine learning. Building machine-learning models relies on minimizing cost functions or maximizing likelihood functions, entropy, and so on in both supervised and unsupervised learning.

Rules for Maxima and Minima for a Univariate Function

- The derivative of *f*(*x*) with respect to *x* would be zero at maxima and minima.
- The second derivative of *f*(*x*), which is nothing but the derivative of the first derivative represented by $\frac{d^2 f(x)}{dx^2}$, needs to be investigated at the point where the first derivative is zero. If the second derivative is less than zero, then it's a point of maxima, while if it is greater than zero it's a point of minima. If the second derivative turns out to be zero as well, then the point is called a point of inflection.

Let's take a very simple function, $y = f(x) = x^2$. If we take the derivative of the function w.r.t *x* and set it to zero, we get $\frac{dy}{dx}=2x=0$, which gives us *x* = 0. Also, the second derivative $\frac{d^2 y}{dx^2}=2$. Hence, for all values of *x*, including *x* = 0, the second derivative is greater than zero and hence *x* = 0 is the minima point for the function *f*(*x*).

Let's try the same exercise for $y = g(x) = x^3$. $\frac{dy}{dx}=3x^2=0$ gives us *x* = 0. The second derivative $\frac{d^2 y}{dx^2}=6x$, and if we evaluate it at *x* = 0 we get 0. So, *x* = 0 is neither the minima nor the maxima point for the function *g*(*x*). Points at which the second derivative is zero are called points of inflection. At points of inflection, the sign of the curvature changes.

The points at which the derivative of a univariate function is zero or the gradient vector for a multivariate function is a zero vector are called stationary points. They may or may not be points of maxima or minima.

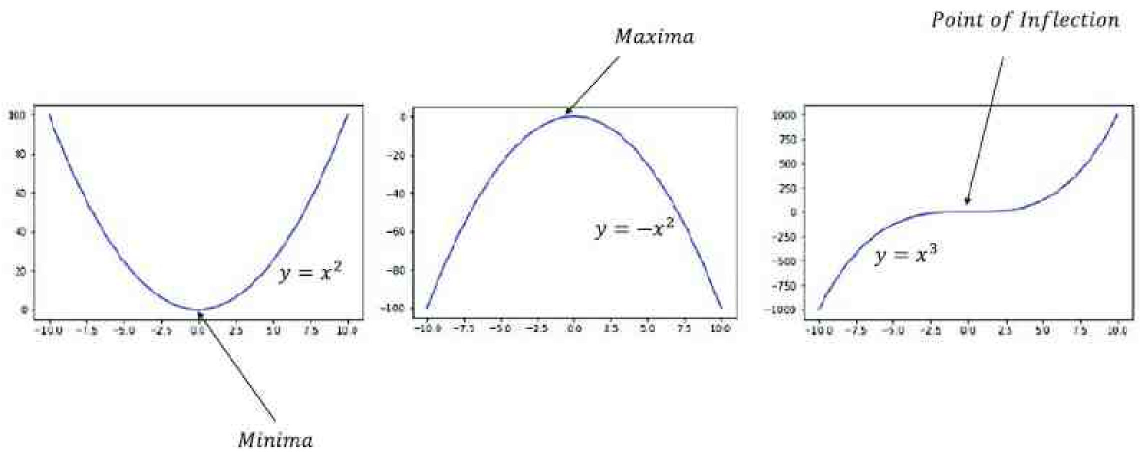


Figure 1-12: Different types of stationary points—maxima, minima, point of inflection

Illustrated in Figure 1-12 are different kinds of stationary points; i.e., maxima, minima, and points of inflection.

Maxima and minima for a multivariate function are a bit more complicated. Let's proceed with an example, and then we will define the rules. We look at a multivariate function with two variables:

$$f(x,y) = x^2y^3 + 3y + x + 5$$

To determine the stationary points, the gradient vector needs to be zero.

$$\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix}^T = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Setting $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ to zero, we get:

$$\frac{\partial f}{\partial x} = 2xy^3 + 1 = 0, \ \frac{\partial f}{\partial x} = 3\text{x}^2\text{y}^2 + 3 = 0$$

We need to compute the Hessian as well:

$$\frac{\partial^2 f}{\partial x^2} = f_{xx} = 2y^3 \ ,$$

$$\frac{\partial^2 f}{\partial y^2} = f_{yy} = 6\text{x}^2 y \, ,$$

$$\frac{\partial^2 f}{\partial x \partial y} = f_{xy} = 6\text{xy}^2$$

$$\frac{\partial^2 f}{\partial y \partial x} = f_{yx} = 6\text{xy}^2$$

For functions with continuous second derivatives, $f_{xy} = f_{yx}$.

Let's say the gradient is zero at ($x = a$, $y = b$):

- If $f_{xx}f_{yy} - (f_{xy})^2 < 0$ *at* ($x = a$, $y = b$) then ($x = a$, $y = b$) is a saddle point.
- If $f_{xx}f_{yy} - (f_{xy})^2 > 0$ *at* ($x = a$, $y = b$) then ($x = a$, $y = b$) is an extremum point; i.e., maxima or minima exists.

  a. If $f_{xx} < 0$ *and* $f_{xy} < 0$ *at* ($x = a$, $y = b$) then $f(x,y)$ has the maximum at ($x = a$, $y = b$).

b. If $f_{xx} > 0$ and $f_{yy} > 0$ at ($x = a$, $y = b$) then $f(x,y)$ has the minimum at ($x = a, y = b$).

- If $f_{xx}f_{yy} - (f_{xy})^2 = 0$ then more advanced methods are required to classify the stationary point correctly.

For a function with n variables, the following are the guidelines for checking for the maxima, minima, and saddle points of a function:

- Computing the gradient and setting it to zero vector would give us the list of stationary points.
- For a stationary point $x_0 \in \mathbb{R}^{n\times 1}$, if the Hessian matrix of the function at x_0 has both positive and negative eigen values, then x_0 is a saddle point. If the eigen values of the Hessian matrix are all positive then the stationarity point is a local minima where as if the eigen values are all negative then the stationarity point is a local maxima.

Local Minima and Global Minima

Functions can have multiple minima at which the gradient is zero, each of which is called a local minima point. The local minima at which the function has the minimum value is called the global minima. The same applies for maxima. Maxima and minima of a function are derived by optimization methods. Since closed-form solutions are not always available or are computationally intractable, the minima and maxima are most often derived through iterative approaches, such as gradient descent, gradient ascent, and so forth. In the iterative way of deriving minima and maxima, the optimization method may get stuck in a local minima or maxima and be unable to reach the global minima or maxima. In iterative methods, the algorithm utilizes the gradient of the function at a point to get to a more optimal point. When traversing a series of points in this fashion, once a point with a zero gradient is encountered, the algorithm stops assuming the desired minima or maxima is reached. This works well when there is a global minima or maxima for the function. Also, the optimization can get stuck at a saddle point too. In all such cases, we would have a suboptimal model.

Illustrated in Figure 1-13 are global and local minima as well as global and local maxima of a function.

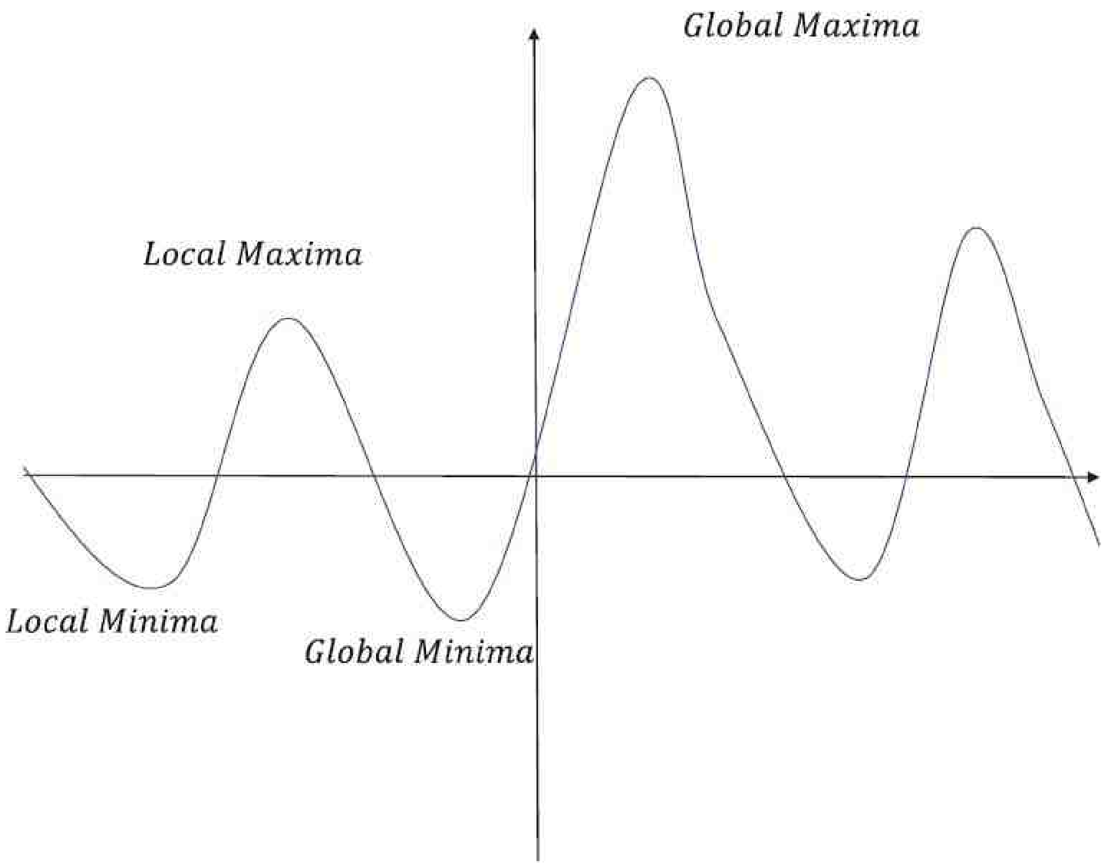


Figure 1-13: Local and global minima/maxima

Positive Semi-Definite and Positive Definite

A square matrix $A \in \mathbb{R}^{n\times n}$ is positive semi-definite if for any non-zero vector $x \in \mathbb{R}^{n\times l}$ the expression $x^TAx \geq 0$. The matrix A is positive definite if the expression $x^T Ax > 0$. All the Eigen values for a positive semi-definite matrix should be non-negative, whereas for a positive definite matrix the Eigen values should be positive. For example, if we consider A as the 2×2 identity matrix—i.e., $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$—then it is positive definite since both of its Eigen values—i.e., 1,1—are positive. Also, if we compute $x^T Ax$, where $x = [x_1\ x_2]^T$, we get $x^T Ax = x_1^2 + x_2^2$, which is always greater than zero for non-zero vector x, which confirms that A is a positive definite matrix.

Convex Set

A set of points is called convex if, given any two points x and y belonging to the set, all points joining the straight line from x to y also belong to the set. In Figure 1-14, a convex set and a non-convex set are illustrated.

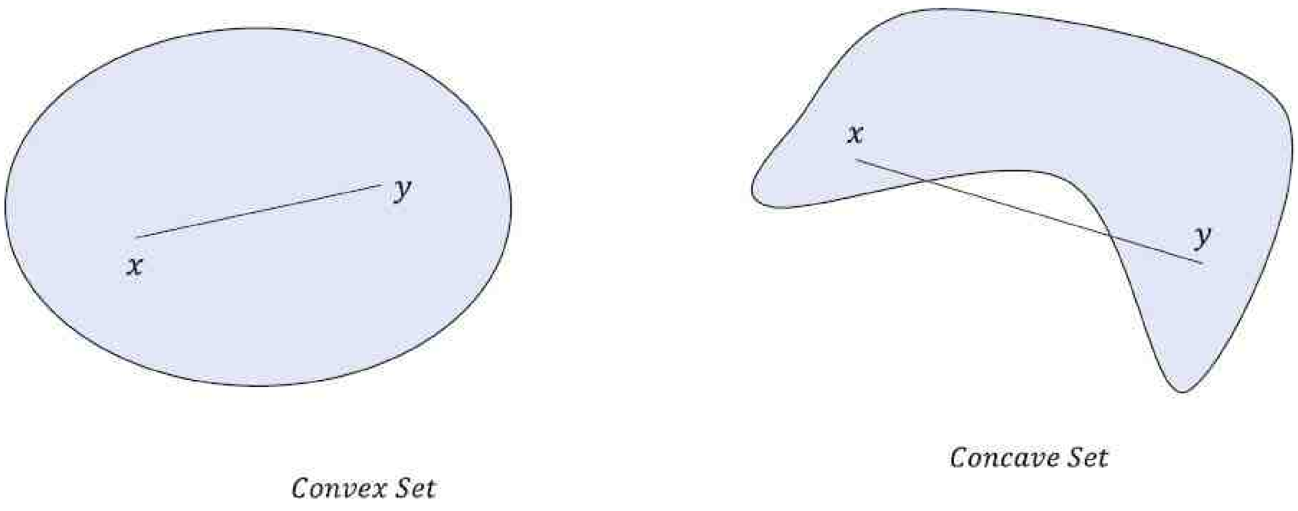


Figure 1-14: Convex and non-convex set

Convex Function

A function $f(x)$ defined on a convex set D, where $x \in \mathbb{R}^{n\times l}$ and D is the domain, is said to be convex if the straight line joining any two points in the function lies above or on the graph of the function. Mathematically, this can be expressed as the following:

$$f(tx+(1-t)y) \le tf(x)+(1-t)f(y) \quad \forall\, x,y \in D, \forall\, t \in [0,1]$$

For a convex function that is twice continuously differentiable, the Hessian matrix of the function evaluated at each point in the domain D of the function should be positive semi-definite; i.e., for any vector $x \in \mathbb{R}^{n\times 1}$,

$$x^T Hx \ge 0$$

A convex function has the local minima as its global minima. Bear in mind that there can be more than one global minima, but the value of the function would be same at each of the global minima for a convex function.

In Figure 1-15, a convex function $f(x)$ is illustrated. As we can see, the $f(x)$ clearly obeys the property of convex functions stated earlier.

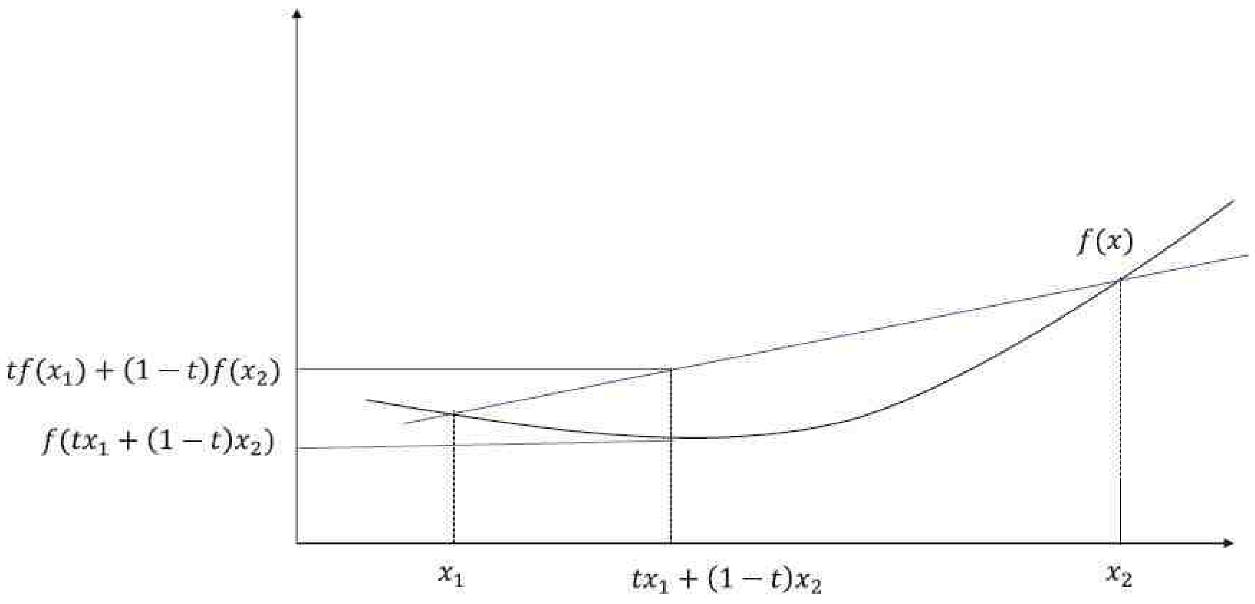


Figure 1-15: Convex function illustration

Non-convex Function

A non-convex function can have many local minima, all of which are not global minima.

In any machine-learning model building process where we try to learn the model parameters by minimizing a cost function, we prefer the cost function to be convex, since with a proper optimization technique we would attain the global minima for sure. For a non-convex cost function, there is a high chance that the optimization technique will get stuck at a local minima or a saddle point, and hence it might not attain its global minima.

Multivariate Convex and Non-convex Functions Examples

Since we would be dealing with high-dimensional functions in deep learning, it makes sense to look at convex and non-convex functions with two variables.

$f(x, y) = 2x^2 + 3y^2 - 5$ is a convex function with minima at $x = 0$, $y = 0$, and the minimum value of $f(x, y)$ at ($x = 0$, $y = 0$) is -5. The function is plotted in Figure 1-16 for reference.

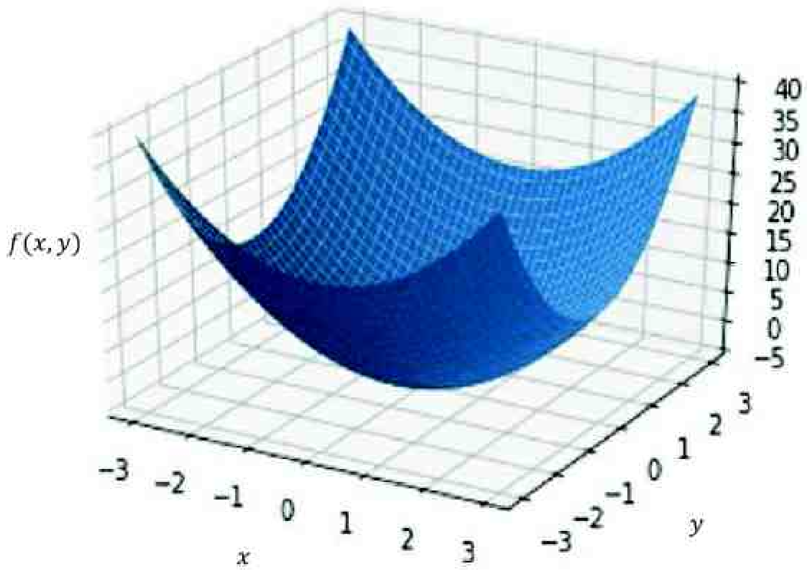


Figure 1-16: Plot of the convex function $2x^2 + 3y^2 - 5$

Now, let's consider the function $f(x,y) = \log x / y \;\forall\; x > 0, y > 0$.

The preceding is a non-convex function, and the easiest way to verify this is to look at the Hessian matrix for the function:

$$\text{Hessian } H = \begin{bmatrix} -\frac{1}{x^2} & 0 \\ 0 & \frac{1}{y^2} \end{bmatrix}$$

The Eigen values of the Hessian are $-\frac{1}{x^2}$ and $\frac{1}{y^2}$. $-\frac{1}{x^2}$ would always be negative for real *x*. Hence, the Hessian is not positive semi-definite, making the function non-convex.

We can see in Figure 1-17 that the plot of the function *log*(*x/y*) looks non-convex.

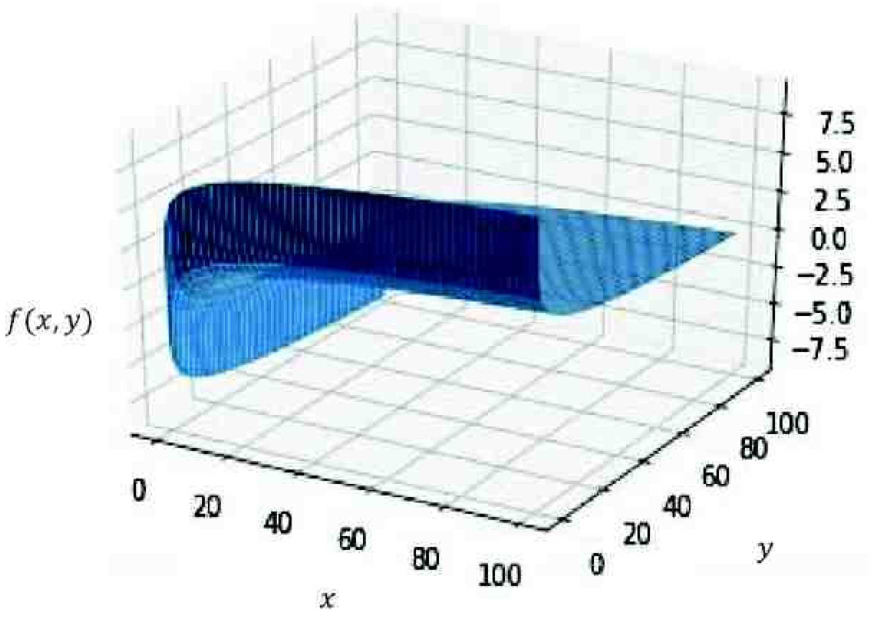


Figure 1-17: Plot of the Non-Convex function log(x/y)

Linear regression through least squares or logistic regression through log-loss cost functions (binary cross entropy) are all convex optimization problems, and hence the model parameters learned through optimization are a global minima solution. Similarly, in SVM the cost function that we optimize is convex.

Whenever there are hidden layers or latent factors involved in any model, the cost function tends to be non-convex in nature. A neural network with hidden layers gives non-convex cost or error surface irrespective of whether we are solving regression or classification problems.

Similarly, in K-means clustering the introduction of clusters makes the cost function to optimize a non-convex cost function. Intelligent methods need to be adopted for non-convex cost functions so that we achieve some local minima that are good enough if it is not possible to reach the global minima.

Parameter initialization becomes very important when dealing with a non-convex problem. The closer the initialized parameters are to the global minima or to some acceptable local minima the better. For *k* means, one method of ensuring that the solution is not suboptimal is to run the *k*-means algorithm several times with different randomly initialized model parameters; i.e., the cluster centroids. We may then take the one that reduces the sum of the intra-cluster variances the most. For neural networks, one needs to use advanced gradient-descent methods involving momentum parameters to come out of the local minima and move forward. We will get to gradient-based optimization methods for neural networks in more detail later in this book.

Taylor Series

Any function can be expressed as an infinite sum by considering the value of the function and its derivatives at a specific point. Such an expansion of the function is called Taylor Series expansion. The Taylor Series expansion of a univariate function around a point *x* can be expressed as follows:

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2!}h^2 f''(x) + \frac{1}{3!}h^3 f'''(x) + .. + \frac{1}{n!}h^n f^n(x) + ..$$

where $f^n(x)$ is the *nth* derivative of the function $f(x)$ and $n!$ denotes the factorial of the number n. The term h has the same dimension as that of x, and both h, x are scalars.

- If $f(x)$ is a constant function, then all the derivatives are zero and $f(x + h)$ and $f(x)$ are same.
- If the function is linear around the neighborhood of x, then for any point $(x + h)$ that lies in the region of the linearity, $f(x + h) = f(x)+hf'(x)$.
- If the function is quadratic around the neighborhood of x, then for any point $(x + h)$ that lies in the quadratic zone, $f(x+h)=f(x)+hf'(x)+\frac{1}{2!}h^2f''(x)$.
- Taylor Series expansion becomes very important in iterative methods such as gradient-descent methods and Newton's methods for optimization as well as in numerical methods for integration and differentiation.

Taylor Series expansion for multivariate functions around a point $x \in \mathbb{R}^{n\times l}$ can be expressed as $f(x+\Delta x)=f(x)+\Delta x^T\nabla f(x)+\frac{1}{2}\Delta x^T\nabla^2 f(x)\Delta x+$ higher order terms

where $\nabla f(x)$ is the gradient vector and $\nabla^2 f(x)$ is the Hessian matrix for the function $f(x)$.

Generally, for practical purposes, we don't go beyond second-order Taylor Series expansion in machine-learning applications since in numerical methods they are hard to compute. Even for second-order expansion computing the Hessian is cost intensive and hence several second-order optimization methods rely on computing the approximate Hessians from gradients instead of evaluating them directly. Please note that the third-order derivatives object $\nabla^3 f(x)$ would be a three-dimensional tensor.

Probability

Before we go on to probability, it is important to know what a random experiment and a sample space are.

In many types of work, be it in a research laboratory or an otherwise, repeated experimentation under almost identical conditions is a standard practice. For example, a medical researcher may be interested in the effect of a drug that is to be launched, or an agronomist might want to study the effect of chemical fertilizer on the yield of a specific crop. The only way to get information about these interests is to conduct experiments. At times, we might not need to perform experiments, as the experiments are conducted by nature and we just need to collect the data.

Each experiment would result in an outcome. Suppose the outcome of the experiments cannot be predicted with absolute certainty. However, before we conduct the experiments, suppose we know the set of all possible outcomes. If such experiments can be repeated under almost the same conditions, then the experiment is called a random experiment, and the set of all possible outcomes is called the sample space.

Do note that sample space is only the set of outcomes we are interested in. A throw of dice can have several outcomes. One is the set of outcomes that deals with the face on which the dice lands. The other possible set of outcomes can be the velocity with which the dice hits the floor. If we are only interested in the face on which the dice lands, then our sample space is $\Omega = \{1, 2, 3, 4, 5, 6\}$; i.e., the face number of the dice.

Let's continue with the experiment of throwing the dice and noting down the face on which it lands as the outcome. Suppose we conduct the experiment n times, and face 1 turns up m times. Then, from the experiment, we can say that the probability of the event of the dice face's being 1 is equal to the number of experiments in which the dice face that turned up was 1 divided by the total number of experiments conducted; i.e., $P(x=1)=\frac{m}{n}$, where x denotes the number on the face of the dice.

Let's suppose we are told that a dice is fair. What is the probability of the number coming up as 1?

Well, given that the dice is fair and that we have no other information, most of us would believe in the near symmetry of the dice such that it would produce 100 faces with number 1 if we were to throw the dice 600 times. This would give us the probability as $\frac{1}{6}$.

Now, let's say we have gathered some 1000 data points about the numbers on the dice head during recent rolls of the dice. Here are the statistics:

$1 \rightarrow 200\ times$

$2 \rightarrow 100\ times$

$3 \rightarrow 100\ times$

$4 \rightarrow 100\ times$

$5 \rightarrow 300\ times$

$6 \rightarrow 200\ times$

In this case, you would come up with the probability of the dice face's being 1 as $P(x = 1) = 200/1000 = 0.2$. The dice is either not symmetric or is biased.

Unions, Intersection, and Conditional Probability

$P(A \cup B)$ = Probability of the event A or event B or both

$P(A \cap B)$ = Probability of event A and event B

$P(A / B)$ = Probability of event A given that B has already occurred.

$$P(A \cap B) = P(A/B)P(B) = P(B/A)P(A)$$

From now on, we will drop the notation of *A* intersection *B* as $A \cap B$ and will denote it as *AB* for ease of notation.

$$P(A - B) = P(A) - P(AB)$$

All the preceding proofs become easy when we look at the Venn Diagram of the two events A and B as represented in Figure 1-18.

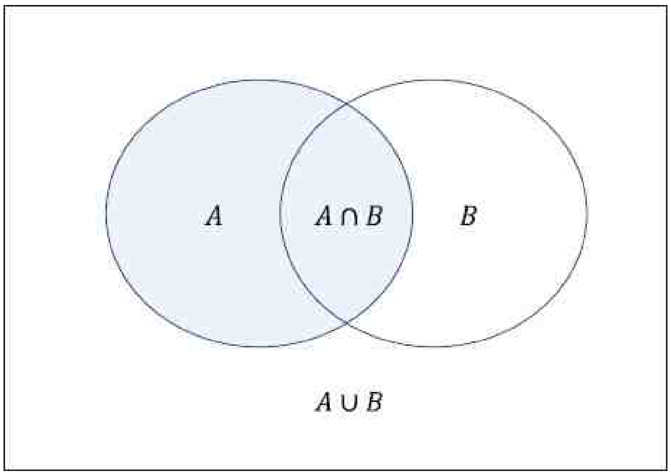


Figure 1-18: Venn diagram of two events A and B showing the union and intersection of the two events

Let's say there are *n* occurrences of an experiment in which *A* has occurred n_1 times, *B* has occurred n_2 times, and *A* and *B* together have occurred *m* times.

Let's represent this with a Venn diagram.

$P(A \cup B)$ can be represented by the sum of the probability of the three disjointed events: $(A - B)$, $(B - A)$, and *AB*.

$$\begin{aligned} P(AUB) &= P(A-B) + P(B-A) + P(AB) \\ &= P(A) - P(AB) + P(B) - P(AB) + P(AB) \\ &= P(A) + P(B) - P(AB) \end{aligned}$$

P(*A/B*) is the probability of *A* given that *B* has already occured. Given than *B* has already happened in n_2 ways, the event *A* is restricted to the event *AB* that can occur in *m* different ways. So, the probability of *A* given *B* can be expressed as follows:

$$P(A/B) = \frac{m}{n_2}$$

Now, $\frac{m}{n_2}$ can be written as $\frac{\frac{m}{n}}{\frac{n_2}{n}} = \frac{P(AB)}{P(B)}$.

Hence, *P*(*A/B*) = *P*(*AB*)/*P*(*B*)=>*P*(*AB*) = *P*(*B*)*P*(*A/B*).

Similarly, if we consider *P*(*B/A*), then the relation *P*(*AB*) = *P*(*A*)*P*(*B/A*) also holds true.

Chain Rule of Probability for Intersection of Event

The product rule of intersection as just discussed for two events can be extended to *n* events.

If A_1, A_2, A_3, …. A_n is the set of *n* events, then the joint probability of these events can be expressed as follows:

$$\begin{aligned} P(A_1A_2A_3\ldots A_n) &= P(A_1)P(A_2/A_1)P(A_3/A_1A_2)\ldots\ldots P\left(A_n/A_1A_2..A_{(n-1)}\right) \\ &= P(A_1)\prod_{i=2}^{n}P\left(A_i/A_1A_2A_3\ldots A_{(n-1)}\right) \end{aligned}$$

Mutually Exclusive Events

Two events *A* and *B* are said to be mutually exclusive if they do not co-occur. In other words, *A* and *B* are mutually exclusive if *P*(*AB*) = 0. For mutually exclusive events, *P*(*A U B*) = *P*(*A*) + *P*(*B*).

In general, the probability of the union of *n* mutually exclusive events can be written as the sum of their probabilities:

$$P(A_1 \cup A_2 .. \cup A_n) = P(A_1) + P(A_2)) + .. P(A_n) = \sum_{i=1}^{n} P(A_i)$$

Independence of Events

Two events *A* and *B* are said to be independent if the probability of their intersection is equal to the product of their individual probabilities; i.e.,

$$P(AB) = P(A)P(B)$$

This is possible because the conditional probability of *A* given *B* is the same as the probability of *A*; i.e.,

$$P(A/B) = P(A)$$

This means that *A* is as likely to happen in the set of all the events as it is in the domain of *B*.

Similarly, *P*(*B / A*) = *P*(*B*) in order for events *A* and *B* to be independent.

When two events are independent, neither of the events is influenced by the fact the other event has happened.

Conditional Independence of Events

Two events *A* and *B* are conditionally independent given a third event *C* if the probability of co-occurrence of *A* and *B* given *C* can be written as follows:

$$P(AB/C) = P(A/C)P(B/C)$$

By the factorization property, *P*(*AB / C*) = *P*(*A / C*)*P*(*B / AC*).

By combining the preceding equations, we see that *P*(*B / AC*) = *P*(*B / C*) as well.

Do note that the conditional independence of events *A* and *B* doesn't guarantee that *A* and *B* are independent too. The conditional independence of events property is used a lot in machine-learning areas where the likelihood function is decomposed into simpler form through the conditional independence assumption. Also, a class of network models known as Bayesian networks uses conditional independence as one of several factors to simplify the network.

Bayes Rule

Now that we have a basic understanding of elementary probability, let's discuss a very important theorem called the Bayes rule. We take two events *A* and *B* to illustrate the theorem, but it can be generalized for any number of events.

We take *P*(*AB*)= *P*(*A*)*P*(*B / A*) from the product rule of probability. (1)

Similarly, *P*(*AB*)= *P*(*B*)*P*(*A/B*). (2)

Combining (1) and (2), we get

$$P(A)P(B/A)=P(B)P(A/B)$$

$$=> P(A/B)=P(A)P(B/A)/P(B)$$

The preceding deduced rule is called the Bayes rule, and it would come handy in many areas of machine learning, such as in computing posterior distribution from likelihood, using Markov chain models, maximizing a posterior algorithm, and so forth.

Probability Mass Function

The probability mass function (pmf) of a random variable is a function that gives the probability of each discrete value that the random variable can take up. The sum of the probabilities must add up to 1.

For instance, in a throw of a fair dice, let the number on the dice face be the random variable *X*.

Then, the pmf can be defined as follows:

$$P(X=i)=\frac{1}{6} \quad i\in\{1,2,3,4,5,6\}$$

Probability Density Function

The probability density function (pdf) gives the probability density of a continuous random variable at each value in its domain. Since it's a continuous variable, the integral of the probability density function over its domain must be equal to 1.

Let *X* be a random variable with domain *D*. *P*(*x*) denotes it's a probability density function, so that

$$\int_D P(x)dx=1$$

For example, the probability density function of a continuous random variable that can take up values from 0 to 1 is given by (*x*) = 2*x* *x* ∈ [0,l]. Let's validate whether it's a probability density function.

For *P*(*x*) to be a probability density function, $\int_{x=0}^{1} P(x)dx$ should be 1.

$\int_{x=0}^{1} P(x)dx=\int_{x=0}^{1} 2xdx=\left[x^2\right]_0^1=1$. Hence, *P*(*x*) is a probability density function.

One thing to be noted is that the integral computes the area under the curve, and since *P*(*x*) is a probability density function (pdf), the area under the curve for a probability curve should be equal to 1.

Expectation of a Random Variable

Expectation of a random variable is nothing but the mean of the random variable. Let's say the random variable *X* takes *n* discrete values, x_1, x_2,….. x_n, with probabilities p_1, p_2, … p_n. In other words, *X* is a discrete random variable with pmf $P(X = x_i) = p_i$. Then, the expectation of the random variable *X* is given by

$$E[X] = x_1p_1 + x_2p_2 + \dots + x_np_n = \sum_{i=1}^{n} x_ip_i$$

If *X* is a continuous random variable with a probability density function of *P*(*x*), the expectation of *X* is given by

$$E[X] = \int_D xP(x)dx$$

where *D* is the domain of *P*(*x*).

Variance of a Random Variable

Variance of a random variable measures the variability in the random variable. It is the mean (expectation) of the squared deviations of the random variable from its mean (or expectation).

Let *X* be a random variable with mean $\mu = E[X]$

$Var[X] = E\left[(X-\mu)^2\right]$ where $\mu = E[X]$

If *X* is a discrete random variable that takes *n* discrete values with a pmf given by $P(X = x_i) = p_i$, the variance of *X* can be expressed as

$$Var[X] = E\left[(X-\mu)^2\right]$$

$$= \sum_{i=1}^{n} (x_i - \mu)^2 p_i$$

If *X* is a continuous random variable having a probability density function of *P*(*x*), then *Var*[*X*] can be expressed as

$$Var[X] = \int_D (x-\mu)^2 P(x)dx$$

where *D* is the domain of *P*(*x*).

Skewness and Kurtosis

Skewness and Kurtosis are higher-order moment statistics for a random variable. Skewness measures the symmetry in a probability distribution, whereas Kurtosis measures whether the tails of the probability distribution are heavy or not. Skewness is a third-order moment and is expressed as

$$Skew(X) = \frac{E[(X-\mu)^3]}{(Var[X])^{3/2}}$$

A perfectly symmetrical probability distribution has a skewness of 0, as shown in the Figure 1-19. A positive value of skewness means that the bulk of the data is toward the left, as illustrated in Figure 1-20, while a negative value of skewness means the bulk of the data is toward the right, as illustrated in Figure 1-21.

Kurtosis is a fourth-order statistic, and for a random variable *X* with a mean of *μ*, it can be expressed as

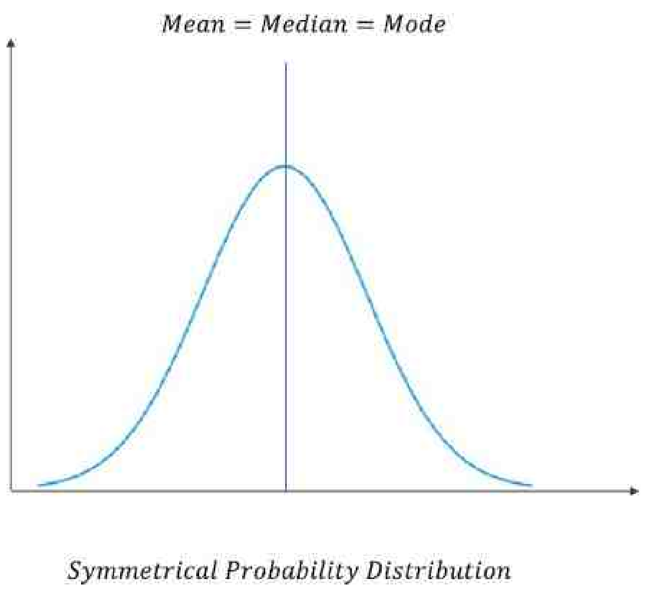


Figure 1-19: Symmetric probability distribution

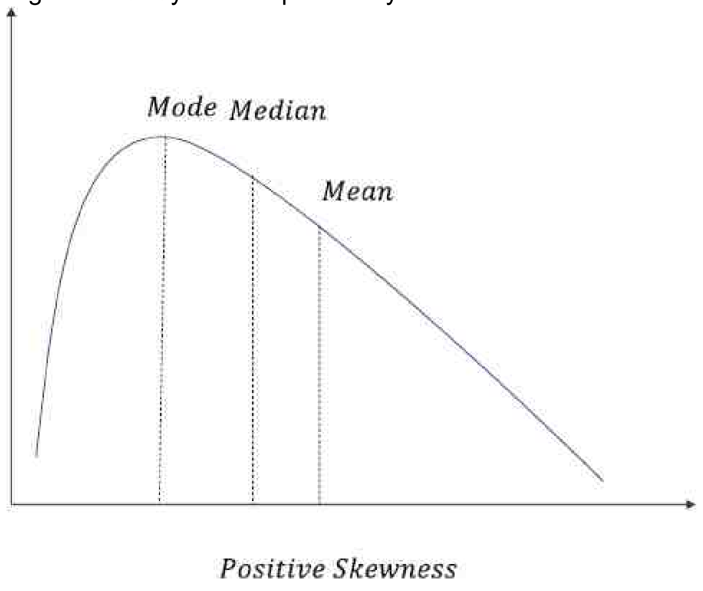


Figure 1-20: Probability distribution with positive skewness

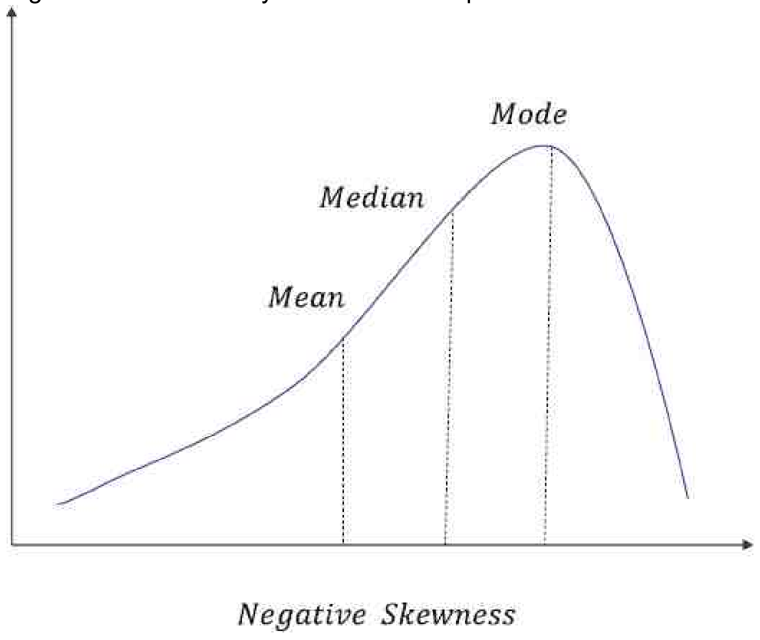


Figure 1-21: Probability distribution with negative skewness

$$Kurt(X) = E\left[[X-\mu]^4\right] / (Var[X])^2$$

Higher Kurtosis leads to heavier tails for a probability distribution, as we can see in Figure 1-23. The Kurtosis for a normal distribution (see Figure 1-22) is 3. However, to measure the Kurtosis of other distributions in terms of a Normal distribution, one generally refers to excess Kurtosis, which is the actual Kurtosis minus the Kurtosis for a normal distribution—i.e., 3.

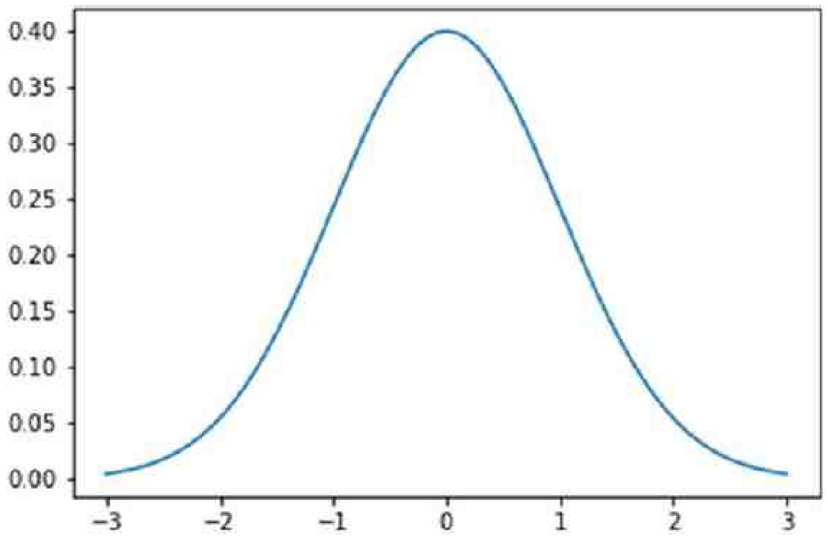


Figure 1-22: Standard normal distribution with Kurtosis = 3

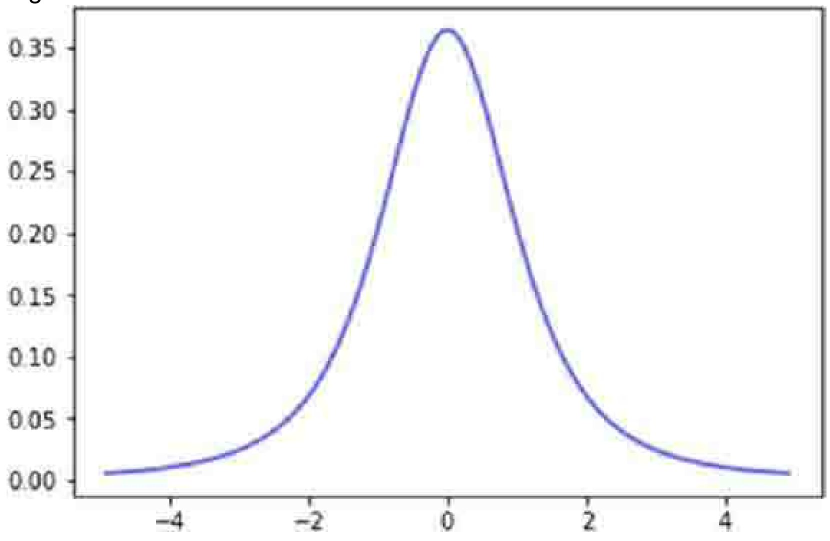


Figure 1-23: Student's T distribution with Kurtosis = ∞

Covariance

The covariance between two random variables *X* and *Y* is a measure of their joint variability. The covariance is positive if higher values of *X* correspond to higher values of *Y* and lower values of *X* correspond to lower values of *Y*. On the other hand, if higher values of *X* correspond to lower values of *Y* and lower values of *X* correspond to higher values of *Y* then the covariance is negative.

The formula for covariance of *X* and *Y* is as follows:

$$cov(X,Y) = E[X-u_x][Y-u_y] \text{ where } u_x = E[X], u_y = E[Y]$$

On simplification of the preceding formula, an alternate is as follows:

$$cov(X,Y) = E[XY] - u_x u_y$$

If two variables are independent, their covariance is zero since $E[XY]=E[X]E[Y]=u_x u_y$

Correlation Coefficient

The covariance in general does not provide much information about the degree of association between two variables, because the two variables maybe on very different scales. Getting a measure of the linear dependence between two variables' correlation coefficients, which is a normalized version of covariance, is much more useful.

The correlation coefficient between two variables *X* and *Y* is expressed as

$$\rho = \frac{cov(X,Y)}{\sigma_x \sigma_y}$$

where σ_X and σ_Y are the standard deviation of *X* and *Y* respectively. The value of *ρ* lies between –1 and +1.

Figure 1-24 illustrates both positive and negative correlations between two variables *X* and *Y*.

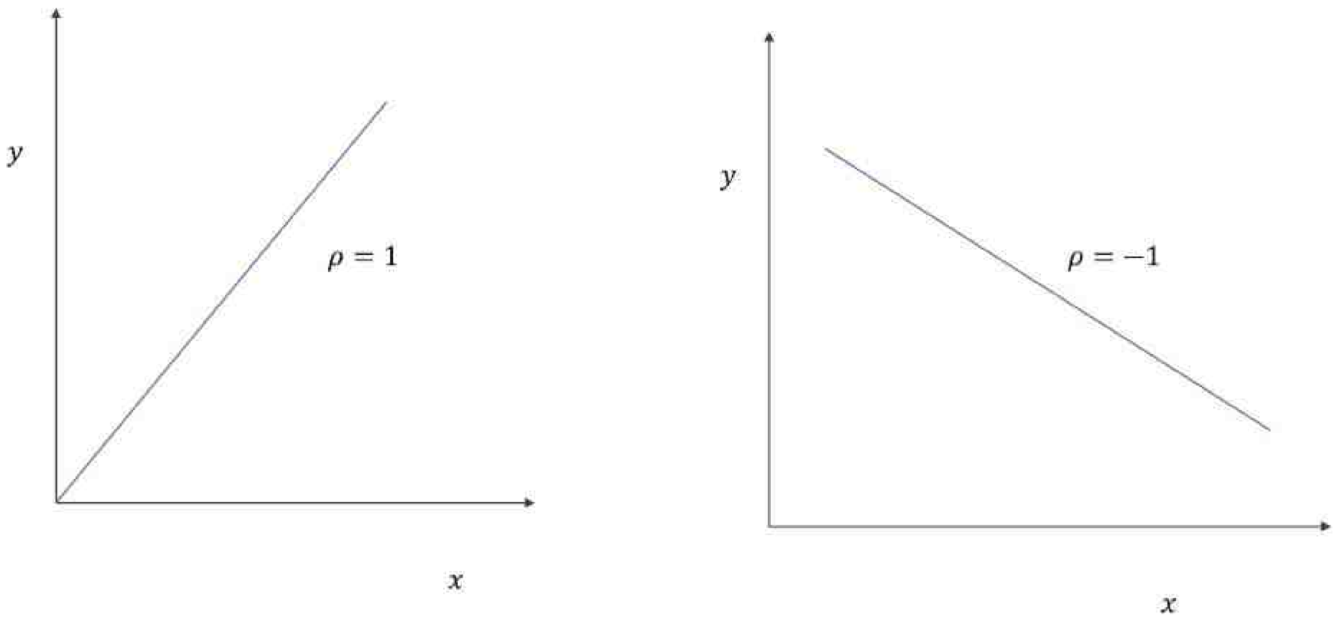


Figure 1-24: Plot of variables with correlation coeffecients of +1 and -1

Some Common Probability Distribution

In this section, we will go through some of the common probability distributions that are frequently used in the machine-learning and deep-learning domains.

Uniform Distribution

The probability density function for a uniform distribution is constant. For a continuous random variable that takes up values between *a* and *b*(*b* > *a*), the probability density function is expressed as

$$P(X=x)=f(x)=\begin{cases} 1/(b-a) & for\ x\in[a,b] \\ 0 & elsewhere \end{cases}$$

Illustrated in Figure 1-25 is the probability density curve for a uniform distribution. The different statistics for a uniform distribution are outlined here:

$$E[X] = \frac{(b+a)}{2}$$

$$Median[X] = \frac{(b+a)}{2}$$

$$Mode[X] = All\ points\ in\ the\ interval\ a\ to\ b$$

$$Var[X] = (b-a)^2/12$$

$$Skew[X] = 0$$

$$Excessive\ Kurt[X] = -6/5$$

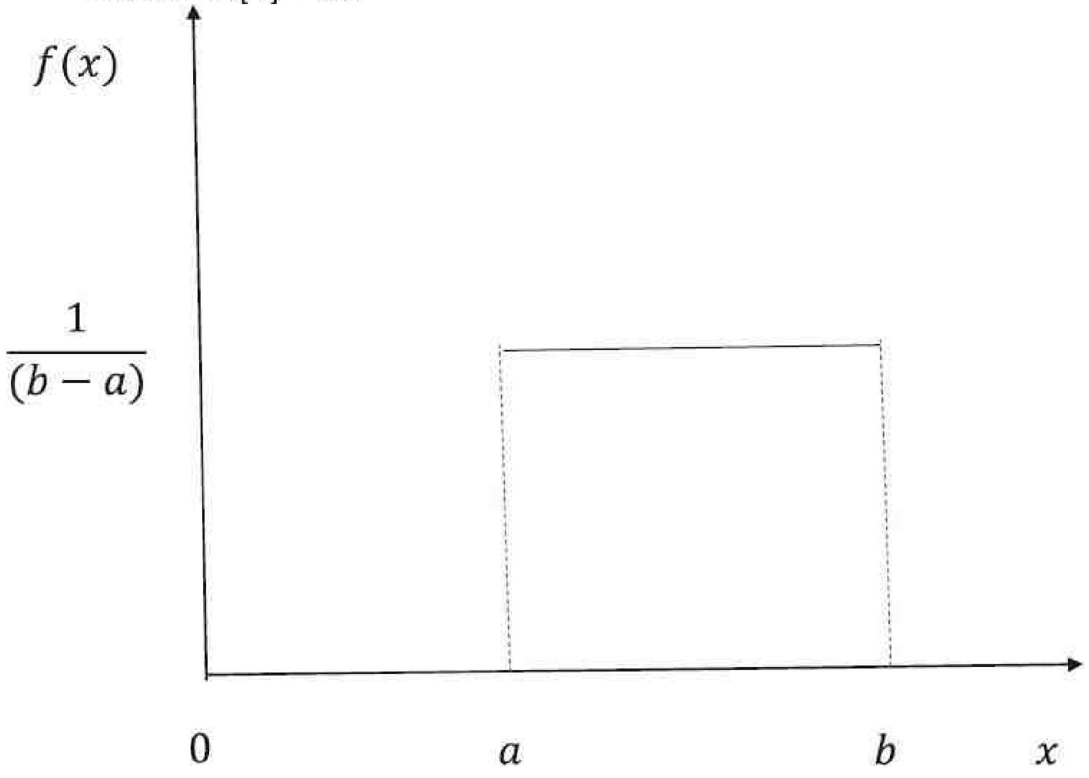


Figure 1-25: Uniform probability distribution

Please note that the excess Kurtosis is the actual Kurtosis minus 3, 3 being the actual Kurtosis for a normal distribution. Hence, the excess Kurtosis is the relative Kurtosis with respect to a normal distribution.

Normal Distribution

This is probably the most important scenario for probability distribution in the real-world. In a normal distribution, the maximum probability density is at the mean of the distribution, and the density falls symmetrically and exponentially to the square of the distance from the mean. The probability density function of a normal distribution can be expressed as

$$P(X=x) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \qquad -\infty < x < +\infty$$

where μ is the mean and σ^2 is the variance of the random variable X. Illustrated in Figure 1-26 is the probability density function of a univariate normal distribution.

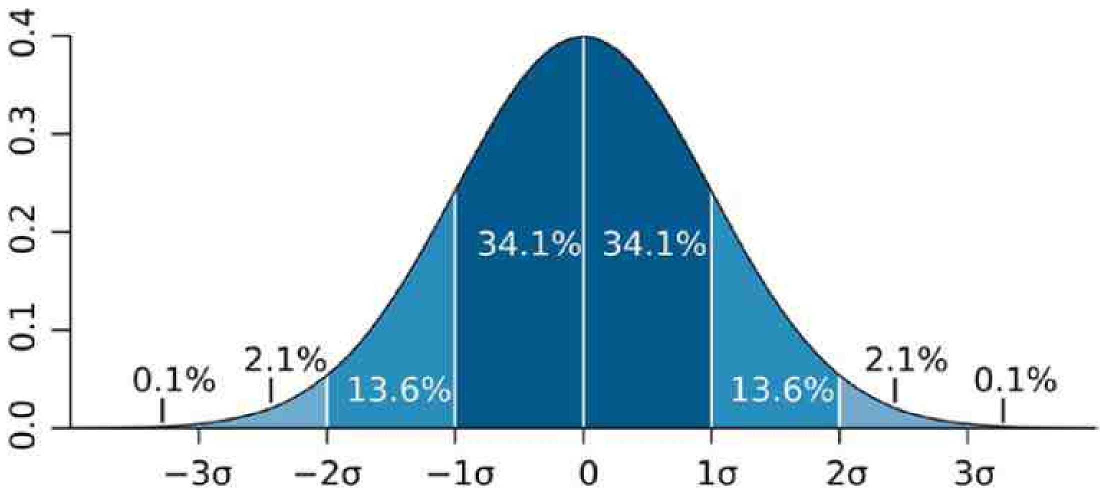


Figure 1-26: Normal probability distribution

As shown in Figure 1-26, 68.2 percent of the data in a normal distribution falls within one standard deviation (+1/-1σ) of the mean, and around 95.4 percent of the data is expected to fall within +2/-2σ of the mean. The important statistics for a normal distribution are outlined here:

$$E[X] = \mu$$

$$Median[X] = \mu$$

$$Mode[X] = \mu$$

$$Var[X] = \sigma^2$$

$$\text{Skew}[X] = 0$$

$$Excess\ Kurt[X] = 0$$

Any normal distribution can be transformed into a standard normal distribution by using the following transformation:

$$z = \frac{(x - \mu)}{\sigma}$$

The mean and standard deviation for the standard normal random variable *z* are 0 and 1 respectively. The standard normal distribution is used a lot in statistical inference tests. Similarly, in linear regression the errors are assumed to be normally distributed.

Multivariate Normal Distribution

A multivariate normal distribution, or Gaussian distribution in *n* variables denoted by vector $x \in \mathbb{R}^{n\times l}$, is the joint probability distribution of the associated variables parameterized by the mean vector $\mu \in \mathbb{R}^{n\times 1}$ and covariance matrix $\Sigma \in \mathbb{R}^{n\times n}$.

The probability density function (pdf) of a multivariate normal distribution is as follows:

$$P(x/\mu;\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{-1/2}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}$$

where $x = [x_1 x_2 \ldots x_n]^T$

$$-\infty < x_i < +\infty \ \ \forall i \in \{1,2,3,..n\}$$

Illustrated in Figure 1-27 is the probability density function of a multivariate normal distribution. A multivariate normal distribution, or Gaussian distribution, has several applications in machine learning. For instance, for multivariate input data that has correlation, the input features are often assumed to follow multivariate normal distribution, and based on the probability density function, points with low probability density are tagged as anomalies. Also, multivariate normal distributions are widely used in a mixture of Gaussian models wherein a data point with multiple features is assumed to belong to several multivariate normal distributions with different probabilities. Mixtures of Gaussians are used in several areas, such as clustering, anomaly detection, hidden Markov models, and so on.

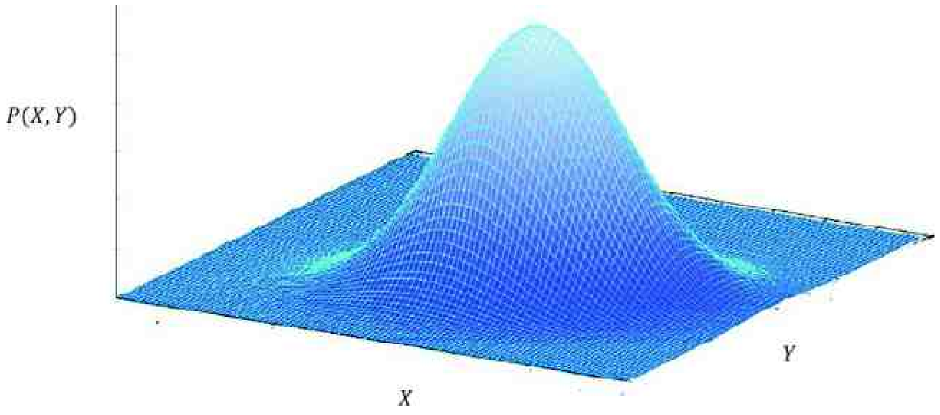


Figure 1-27: Multivariate normal distribution in two variables

Bernoulli Distribution

An experiment in which the two outcomes are mutually exclusive and exhaustive (the sum of probability of the two outcomes is 1) is called a Bernoulli trail.

A Bernoulli trail follows a Bernoulli distribution. Let's say in a Bernoulli trail the two outcomes are success and failure. If the probability of success is *p* then, since these two events exhaust the sample space, the probability of failure is 1 – *p*. Let *x* = 1 denote success. Thus, the probability of success or failure can be denoted as follows:

$$P(X=x)= f(x)=p^x(1-p)^{(1-x)} \qquad x\in\{0,1\}$$

The preceding expression for *P*(*X* = *x*) denotes the probability mass function of a Bernoulli distribution. The expectation and variance of the probability mass function are as follows:

$$E[X]= p$$

$$Var[X]=p(1-p)$$

The Bernoulli distribution can be extended to multiclass events that are mutually exclusive and exhaustive. Any two-class classification problem can be modeled as a Bernoulli trail. For instance, the logistic regression likelihood function is based on a Bernoulli distribution for each training data point, with the probability *p* being given by the sigmoid function.

Binomial Distribution

In a sequence of Bernoulli trails, we are often interested in the probability of the total number of successes and failures instead of the actual sequence in which they occur. If in a sequence of *n* successive Bernoulli trails *x* denotes the number of successes, then the probability of *x* successes out of *n* Bernoulli trails can be expressed by a probability mass function denoted by

$$P(X=x)=\binom{n}{x}p^x(1-p)^{(n-x)} \qquad x\in\{0,1,2\ldots,n\}$$

where *p* is the probability of success.

The expectation and variance of the distribution are as follows:

$$E[X]=np$$

$$Var[X]=np(1-p)$$

Illustrated in Figure 1-28 is the probability mass function of a binomial distribution with *n* = 4 and *p* = 0.3.

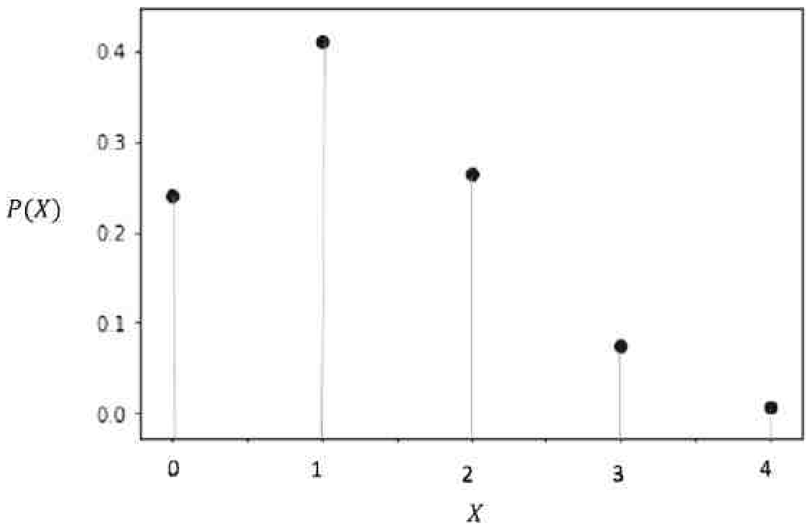


Figure 1-28: Probability Mass function of a Binomial Distribution with n=4 and p = 0.3

Poisson Distribution

Whenever the rate of some quantity is of concern, like the number of defects in a 1000-product lot, the number of alpha particles emitted by a radioactive substance in the previous four-hour duration, and so on, Poisson distribution is generally the best way to represent such phenomenon. The probability mass function for Poisson distribution is as follows:

$$P(X=x)=\frac{e^{-\lambda}\lambda^{x}}{x!} \quad where\ x\in\{0,1,2,\ldots\ldots\ldots\infty\}$$

$$E[X]=\lambda$$

$$Var[X]=\lambda$$

Illustrated in Figure 1-29 is the probability mass function of a Poisson distribution with mean of λ = 15.

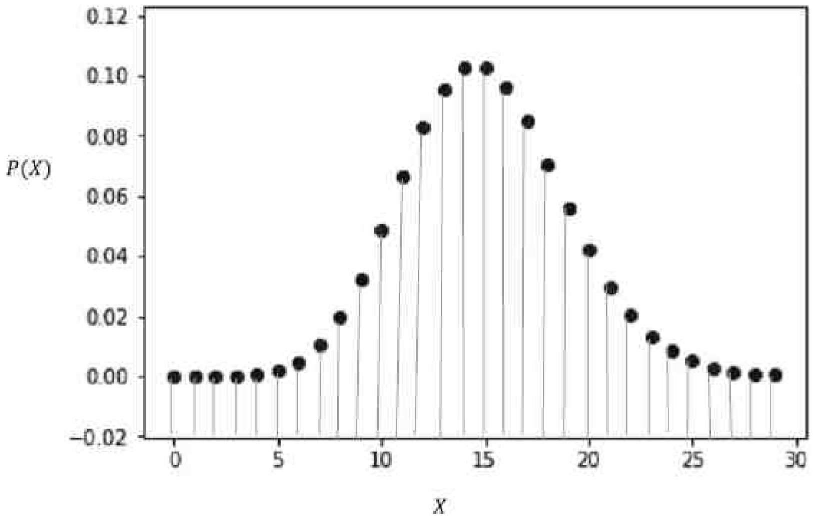


Figure 1-29: Probability mass function of a Poisson distribution with mean = 15

Likelihood Function

Likelihood is the probability of the observed data given the parameters that generate the underlying data. Let's suppose we observe *n* observations x_1, x_2,.....x_n and assume that the observations are independent and identically normally distributed with mean μ and variance σ^2.

The likelihood function in this case would be as follows:

$$P(Data\,/\,Model\ parameters) = P(x_1, x_2, \ldots\ldots x_n\,/\,\mu, \sigma^2\,)$$

Since the observations are independent, we can factorize the likelihood as follows:

$$P(Data\,/\,Model\ parameters) = \prod_{i=1}^{n} P(x_i\,/\,\mu, \sigma^2\,)$$

Each of the x_i ~ *Normal*(μ,σ^2), hence the likelihood can be further expanded as follows:

$$P(Data\,/\,Modelparameters) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-(x_i-\mu)^2}{2\sigma^2}}$$

Maximum Likelihood Estimate

Maximum likelihood estimate (MLE) is a technique for estimating the parameters of a distribution or model. This is achieved by deriving the parameters that would maximize the likelihood function—i.e., maximize the probability of observing the data given the parameters of the model. Let's work through an example to understand maximum likelihood estimates.

Suppose Adam tosses a coin 10 times and observes 7 heads and 33tails. Also, assume that the tosses are independent and identical. What would be the maximum likelihood estimate for the probability of heads for the given coin?

Each toss of a coin is a Bernoulli trial, with the probability of heads being, let's say, p, which is an unknown parameter that we want to estimate. Also, let the event that a toss produces heads be denoted by 1 1and tails by 0.

The likelihood function can be represented as follows:

$$P(Data\,/\,parameter) = L(p) = P(x_1, x_2, \ldots\ldots x_{10}\,/\,p\,)$$

$$= \prod_{i=1}^{10} P(x_i\,/\,p\,)$$

$$= p^7(1-p)^3$$

Just for clarification, let's see how the likelihood *L* came to be $p^7(1 - p)^3$.

For each heads, the probability from the Bernoulli distribution is $P(x_i = 1 / p) = p^1\ (1 - p)^0 = p$. Similarly, for each tails the probability is $P(x_i = 0 / p) = p^0\ (1 - p)^1 = 1 - p$. As we have 7 heads and 3 tails, we get the likelihood $L(p)$ to be $p^7(1 - p)^3$.

To maximize the likelihood *L*, we need to take the derivate of *L* with respect to *p* and set it to 0.

Now, instead of maximizing the likelihood *L*(*p*) we can maximize the logarithm of the likelihood—i.e., *logL*(*p*). Since logarithmic is a monotonically increasing function, the parameter value that maximizes *L*(*p*) would also maximize *logL*(*p*). Taking the derivative of the log of the likelihood is mathematically more convenient than taking the derivative of the product form of the original likelihood.

$$logL(p) = 7logp + 3log(1-p)$$

Taking the derivative of both sides and setting it to zero looks as follows:

$$\frac{dLog(L(p))}{dp} = \frac{7}{p} - \frac{3}{1-p} = 0$$

$$\Rightarrow p = 7/10$$

Interested readers can compute the second derivative $\frac{d^2\log(L)}{dp^2}$ at $p = \frac{7}{10}$; you will for sure get a negative value, confirming that $p = \frac{7}{10}$ is indeed the point of maxima.

Some of you would have already had $\frac{7}{10}$ in mind without even going though maximum likelihood, just by the basic definition of probability. As you will see later, with this simple method a lot of complex model parameters are estimated in the machine-learning and deep-learning world.

Let's look at another little trick that might come in handy while working on optimization. Computing the maxima of a function $f(x)$ is the same as computing the minima for the function $-f(x)$. The maxima for $f(x)$ and the minima for $-f(x)$ would take place at the same value of x. Similarly, the maxima for $f(x)$ and the minima for $1/f(x)$ would happen at the same value of x.

Often in machine-learning and deep-learning applications we use advanced optimization packages, which only know to minimize a cost function in order to compute the model parameters. In such cases, we conveniently convert the maximization problem to a minimization problem by either changing the sign or taking the reciprocal of the function, whichever makes more sense. For example, in the preceding problem we could have taken the negative of the log likelihood function—i.e., *LogL*(*p*)—and minimized it; we would have gotten the same probability estimate of 0.7.

Hypothesis Testing and p Value

Often, we need to do some hypothesis testing based on samples collected from a population. We start with a null hypothesis and, based on the statistical test performed, accept the null hypothesis or reject it.

Before we start with hypothesis testing, let's first consider one of the core fundamentals in statistics, which is the Central Limit theorem.

Let $x_1, x_2, x_3, \ldots \cdot \cdot, x_n$ be the n independent and identically distributed observation of a sample from a population with mean μ and finite variance σ^2.

The sample mean denoted by $\bar{x}$ follows normal distribution, with mean μ and variance $\frac{\sigma^2}{n}$; i.e.,

$$\bar{x} \sim Normal\left(\mu, \frac{\sigma^2}{n}\right) \text{ where } \bar{x} = \frac{x_1 + x_2 + x_3 + \ldots + x_n}{n}$$

This is called the Central Limit theorem. As the sample size n increases, the variance of $\bar{x}$ reduces and tends toward zero as $n \to \infty$.

Figure 1-30 illustrates a population distribution and a distribution of the mean of samples of fixed size n drawn from the population.

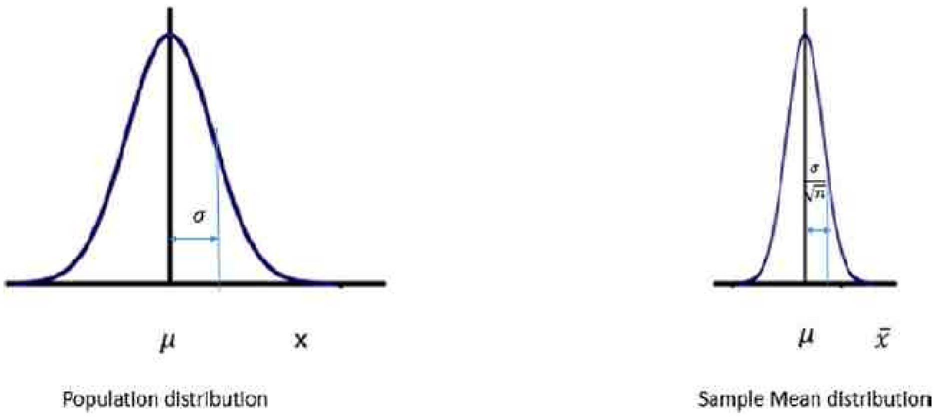


Figure 1-30: Distribution for population and distribution for sample mean

Please note that the sample mean follows normal distribution irrespective of whether the population variable is normally distributed or not. Now, let's consider a simple hypothesis-testing problem.

Boys who are 10 years old are known to have a mean weight of 85 pounds, with a standard deviation of 11.6. Boys in one county are checked as to whether they are obese. To test this, the mean weight of a group of 25 random boys from the county is collected. The mean weight is found to be 89.16 pounds.

We would have to form a null hypothesis, and would reject it through the test if the evidence against the null hypothesis were strong enough.

Let us consider the null hypothesis: H_0. The children in the county are not obese; i.e., they come from the same population with a mean of $\mu = 85$.

Under the null hypothesis H_0, the sample mean is as follows:

$$\bar{x} \sim Normal\left(85, \frac{11.6^2}{25}\right)$$

The closer the sample mean observed is to the population mean, the better it is for the null hypothesis to be true. On the other hand, the further the sample mean observed is away from the population mean, the stronger the evidence is against the null hypothesis.

The standard normal variate $z = (\bar{x} - \mu)/(\sigma / \sqrt{n}) = (89.16 - 85)/(11.6/\sqrt{25}) = +1.75$

For every hypothesis test, we determine a *p* value. The *p* value of this hypothesis test is the probability of observing a sample mean that is further away from what is observed; i.e., $P(\bar{x} \geq 89.16)$ or $P(z \geq 1.75)$. So, the smaller the *p* value is, the stronger the evidence is against the null hypothesis.

When the *p* value is less than a specified threshold percentage α, which is called the type-1 error, the null hypothesis is rejected.

Please note that the deviation of the sample mean from the population can be purely the result of randomness since the sample mean has finite variance σ^2/n. The α gives us a threshold beyond which we should reject the null hypothesis even when the null hypothesis is true. We might be wrong, and the huge deviation might just be because of randomness. But the probability of that happening is very small, especially if we have a large sample size, since the sample mean standard deviation reduces significantly. When we do reject the null hypothesis even if the null hypothesis is true, we commit a type-I error, and hence α gives us the probability of a type-1 error.

The *p* value for this test is $P(Z \geq 1.75) = 0.04$

The type-I error α that one should choose depends on one's knowledge of the specific domain in which the test is performed. Generally, $\alpha = 0.05$ is a good enough type-1 error setting. Since the *p* value computed is less than the type-I error specified for the test, we cannot accept the null hypothesis. We say the test is statistically significant. The *p* value has been illustrated in Figure 1-31.

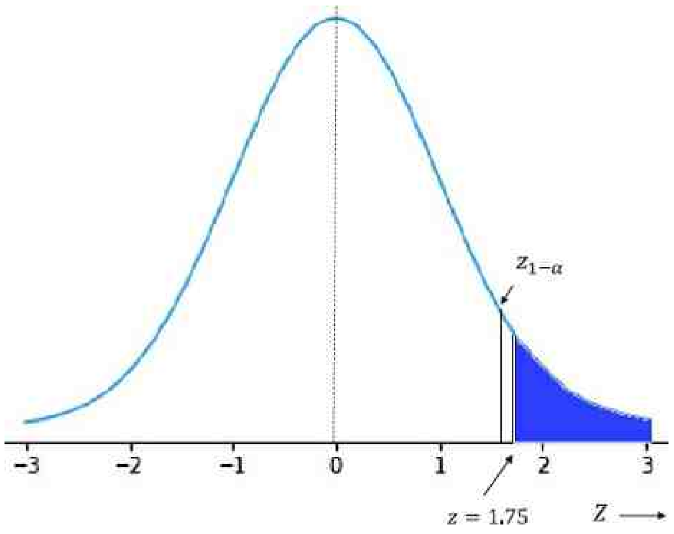


Figure 1-31: Z test showing p value

The dark-colored area corresponds to the *p* value; i.e., $P(z \geq 1.75)$. $Z_{1-\alpha}$ corresponds to the *z* value beyond which we are likely to commit a Type-1 error given the null hypothesis is true. The area beyond $z_{l-\alpha}$ —i.e., $P(z \geq Z_{1-\alpha})$—stands for the Type-1 error probability. Since the *p* value is less than the Type-1 error probability for the test, the null hypothesis can't be taken as true. A *Z* test such as this is generally followed up by another good practice—the Confidence Interval test.

Also, the preceding test, popularly known as the *Z* test. is not always possible unless we have the population variance provided to us. For certain problems, we might not have the population variance. In such cases, the Student-T test is more convenient since it uses sample variances instead of population variances.

The reader is encouraged to explore more regarding these statistical tests.

Formulation of Machine-Learning Algorithm and Optimization Techniques

The aim of modeling is to minimize the cost function of the model parameters given the data by using different optimization techniques. One may ask that if we set the derivative or gradient of the cost function to zero would we have the model parameters. This is not always possible, since all solutions might not have a closed-form solution, or the closed-form solution might be computationally expensive or intractable. Further, when the data size is huge there would be memory constraints when going for a closed-form solution. Hence, iterative methods are generally used for complex optimization problems.

Machine learning can be broadly classified into two types:

- Supervised machine learning
- Unsupervised machine learning

Supervised Learning

In supervised learning, each training data point is associated with several input features—typically an input feature vector and its corresponding label. A model is constructed with several parameters that try to predict the output label given the input feature vector. The model parameters are derived by optimizing some form of cost function that is based on the error of prediction; i.e., the discrepancy between the actual labels and the predicted labels for the training data points. Alternatively, maximizing the likelihood of the training data would also provide us with the model parameters.

Linear Regression as a Supervised Learning Method

We might have a dataset that has the prices for houses as the target variable or output label, whereas features like area of the house, number of bedrooms, number of bathrooms, and so forth are its input feature vector. We can define a function that would predict the price of the house based on the input feature vector.

Let the input feature vector be represented by x' and the predicted value be y_p. Let the actual value of the housing price—i.e., the output label—be denoted by y. We can define a model where the output label is expressed as a function of the input feature vector, as shown in the following equation. The model is parameterized by several constants that we wish to learn via the training process.

$$y / x' = \theta'^T x' + b + \epsilon$$

where ϵ is the random variation in prediction and $\epsilon \sim Normal(0,\sigma^2)$.

So, the housing price given an input (y / x') is a linear combination of the input vector x' plus a bias term b and a random component ϵ, which follows a normal distribution with a 0 mean and a finite variance of σ^2

As ϵ is a random component, it can't be predicted, and the best we can predict is the mean of housing prices given a feature value i.e.

The predicted value $y_p / x' = E[y / x'] = \theta'^T x' + b$

Here, θ' is the linear combiner and b is the bias or the intercept. Both θ' and b are the model parameters that we wish to learn. We can express $y_p = \theta^T x$, where the bias has been added to the model parameter corresponding to the constant feature 1. This small trick makes the representation simpler.

Let's say we have m samples $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$.... $(x^{(m)}, y^{(m)})$. We can compute a cost function that takes the sum of the squares of the difference between the predicted and the actual values of the housing prices and try to minimize it in order to derive the model parameters.

The cost function can be defined as

$$C(\theta) = \sum_{i=1}^{m} \left(\theta^T x_i - y^{(i)}\right)^2$$

We can minimize the cost function with respect to θ to determine the model parameter. This is a linear regression problem where the output label or target is continuous. Regression falls under the supervised class of learning. Figure 1-32 illustrates the relationship between housing prices and number of bedrooms.

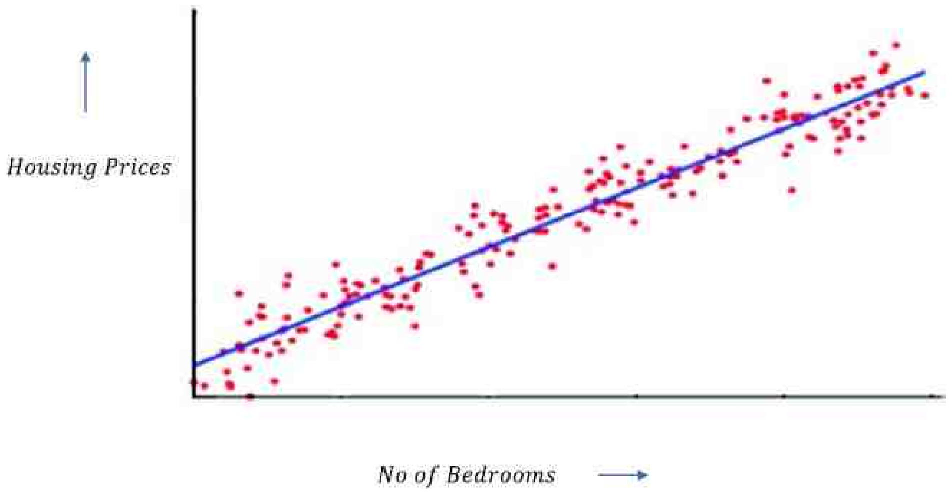


Figure 1-32: Regression fit to the Housing Prices versus Number of Bedrooms data. The red points denote the data points, and the blue line indicates the fitted regression line.

Let the input vector be $x' = [x_1 x_2\ x_3]^T$, where

$x_1 \rightarrow \textit{the area of the house}$

$x_2 \rightarrow \textit{the number of bedrooms}$

$x_3 \rightarrow \textit{the number of bathrooms}$

Let the parameter vector corresponding to the input feature vector be $\theta' = [\theta_1\ \theta_2\ \theta_3]^T$, where

$\theta_1 \rightarrow \textit{additional Cost per unit area}$

$\theta_2 \rightarrow \textit{additional cost per bedroom}$

$\theta_3 \rightarrow \textit{additional cost per bathroom}$

After taking into consideration the bias term, the input feature vector becomes $x = [x_0\ x_1\ x_2\ x_3]^T$, where

$x_0 \rightarrow \textit{constant value of } 1, \textit{i.e., feature corresponding to the bias term}$

$x_1 \rightarrow \textit{the area of the house}$

$x_2 \rightarrow \textit{the number of bedrooms}$

$x_3 \rightarrow \textit{the number of bathrooms}$

and $\theta = [\theta_0 \theta_1 \theta_2 \theta_3]$, where

$$\theta_0 \rightarrow bias\ term\ or\ intercept$$

$$\theta_1 \rightarrow additional\ cost\ per\ unit\ area$$

$$\theta_2 \rightarrow additional\ cost\ \ per\ bedroom$$

$$\theta_3 \rightarrow additional\ cost\ per\ bathroom$$

Now that we have some understanding of how to construct a regression problem and its associated cost function, let's simplify the problem and proceed toward deriving the model parameters.

Model parameter $\theta^* = \underbrace{Arg\ Min}_{\theta} C(\theta) = \underbrace{\text{Arg Min}}_{\theta} \sum_{i=1}^{m} \left(\theta^T x_i - y^{(i)}\right)^2$

The input vectors for all the samples can be combined to a matrix *X*, and the corresponding target output can be represented as a vector *Y*.

$$X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)}\, x_2^{(1)} x_3^{(1)} \\ x_0^{(2)} & x_1^{(2)}\, x_2^{(2)} x_3^{(2)} \\ x_0^{(3)} & x_1^{(3)}\, x_2^{(3)} x_3^{(3)} \\ & . \\ & . \\ x_0^{(m)} & x_1^{(m)}\, x_2^{(m)} x_3^{(m)} \end{bmatrix} Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ . \\ . \\ y^{(m)} \end{bmatrix}$$

If we represent the vector of predictions as *Yp*, then *Yp* = *Xθ*. So, the error in prediction vector *e* can be represented as follows:

$$e = X\theta - Y$$

Hence, *C(θ)* can be expressed as the square of the l^2 norm of the error vector *e*, i.e.,

$$\begin{aligned} C(\theta) &= \|e\|_2^2 \\ &= \|X\theta - Y\|_2^2 \\ &= (X\theta - Y)^T (X\theta - Y) \end{aligned}$$

Now that we have a simplified cost function in matrix form, it will be easy to apply different optimization techniques. These techniques would work for most cost functions, be it a convex cost function or a non-convex one. For non-convex ones, there are some additional things that need to be considered, which we will discuss in detail while considering neural networks.

We can directly derive the model parameters by computing the gradient and setting it to zero vector. You can apply the rules that we learned earlier to check if the conditions of minima are satisfied.

The gradient of the cost function with respect to the parameter vector *θ* is as seen here:

$$\nabla C(\theta) = 2X^T (X\theta - Y)$$

Setting $\nabla C(\theta) = 0$, we get $X^T X\theta = X^T Y \Rightarrow \hat{\theta} = \left(X^T X\right)^{-1} X^T Y$.

If one looks at this solution closely one can observe that the pseudo-inverse of *X*—i.e., $(X^T X)^{-1} X^T$ — comes into the solution of the linear regression problem. This is because a linear regression parameter vector can be looked at as a solution to the equation $X\theta = Y$, where *X* is an $m \times n$ rectangular matrix with $m > n$.

The preceding expression for $\hat{\theta}$ is the closed-form solution for the model parameter. Using this derived $\hat{\theta}$ for new data point x_{new}, we can predict the price of housing as $\hat{\theta}^T x_{new}$.

The computation of the inverse of (X^TX) is both cost and memory intensive for large datasets. Also, there are situations when the matrix X^TX is singular and hence its inverse is not defined. Therefore, we need to look at alternative methods to get to the minima point.

One thing to validate after building a linear regression model is the distribution of residual errors for the training data points. The errors should be approximately normally distributed with a 0 mean and some finite variance. The QQ plot that plots the actual quantile values for the error distribution versus the theoretical quantile values for the error distribution can be used to check whether the assumption of Gaussianity for the residual is satisfied.

Linear Regression Through Vector Space Approach

The linear regression problem is to determine the parameter vector θ such that $X\theta$ is as close to the output vector Y as possible. $X \in \mathbb{R}^{m\times n}$ is the data matrix and can be thought of as n column vectors $c_i \in \mathbb{R}^{m\times 1}$ stacked one after the other, as shown here:

$$X = \left[c_1 c_2 \ldots . c_n\right]$$

The dimension of column space is m whereas the number of column vectors is n. Hence, the column vectors at best can span a subspace of dimension n within the m-dimensional vector space. The subspace is depicted in Figure 1-33. Although it looks like a 2-D surface, we need to imagine it as an n-dimensional space. The linear combination of the columns of X with the parameters gives the prediction vector, as shown here:

$$X\theta = \theta_1 c_1 + \theta_1 c_1 + ..\theta_n c_n$$

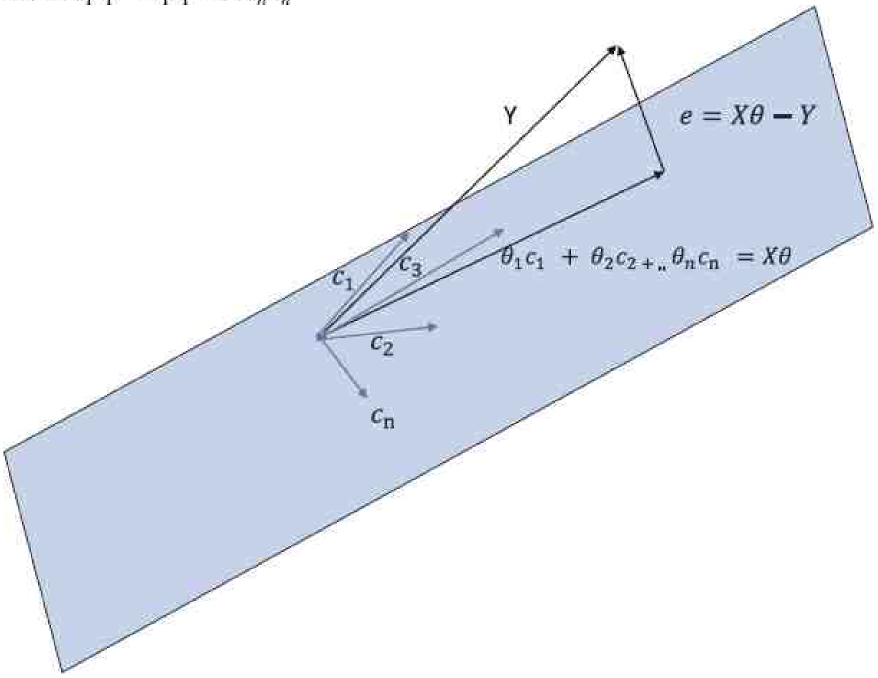


Figure 1-33: Linear regression as a vector space problem

Since $X\theta$ is nothing but the linear combination of the column vectors of X, $X\theta$ stays in the same subspace as the ones spanned by the column vectors c_i $\forall i = \{1,2,3\ldots n\}$.

Now the actual target value vector Y lies outside the subspace spanned by the column vectors of X, thus no matter what θ we combine X with, $X\theta$ can never equal or align itself in the direction of Y. There is going to be a non-zero error vector given by $e = Y - X\theta$.

Now that we know that we have an error, we need to investigate how to reduce the l^2 norm of the error. For the l^2 norm of the error vector to be at a minimum it should be perpendicular to the prediction vector $X\theta$. Since $e = Y - X\theta$ is perpendicular to $X\theta$ it should be perpendicular to all vectors in that subspace.

So, the dot product of all the column vectors of X with the error vector $Y - X\theta$ should be zero, which gives us the following:

$$c_1{}^T[Y - X\theta] = 0, c_2{}^T[Y - X\theta] = 0, \ldots c_n{}^T[Y - X\theta] = 0$$

This can be rearranged in a matrix form as follows:

$$[c_1 c_2 \ldots c_n]^T [Y - X\theta] = 0$$

$$\Rightarrow X^T[Y - X\theta] = 0 \Rightarrow \hat{\theta} = (X^T X)^{-1} X^T Y$$

Also, please note that the error vector will be perpendicular to the prediction vector only if *Xθ* is the projection of *Y* in the subspace spanned by the column vectors of *X*. The sole purpose of this illustration is to emphasize the importance of vector spaces in solving machine-learning problems.

Classification

Similarly, we may look at classification problems where instead of predicting the value of a continuous variable we predict the class label associated with an input feature vector. For example, we can try to predict whether a customer is likely to default based on his recent payment history and transaction details as well as his demographic and employment information. In such a problem, we would have data with the features just mentioned as input and a target indicating whether the customer has defaulted as the class label. Based on this labelled data, we can build a classifier that can predict a class label indicating whether the customer will default, or we can provide a probability score that the customer will default. In this scenario, the problem is a binary classification problem with two classes—the defaulter class and the non-defaulter class. When building such a classifier, the least square method might not give a good cost function since we are trying to guess the label and not predict a continuous variable. The popular cost functions for classification problems are generally log-loss cost functions that are based on maximum likelihood and entropy-based cost functions, such as Gini Entropy and Shannon Entropy.

The classifiers that have linear decision boundaries are called linear classifiers. Decision boundaries are hyperplanes or surfaces that separate the different classes. In the case of linear decision boundaries, the separating plane is a hyperplane.

Figures 1-34 and 1-35 illustrate linear and non-linear decision boundaries respectively for the separation of two classes.

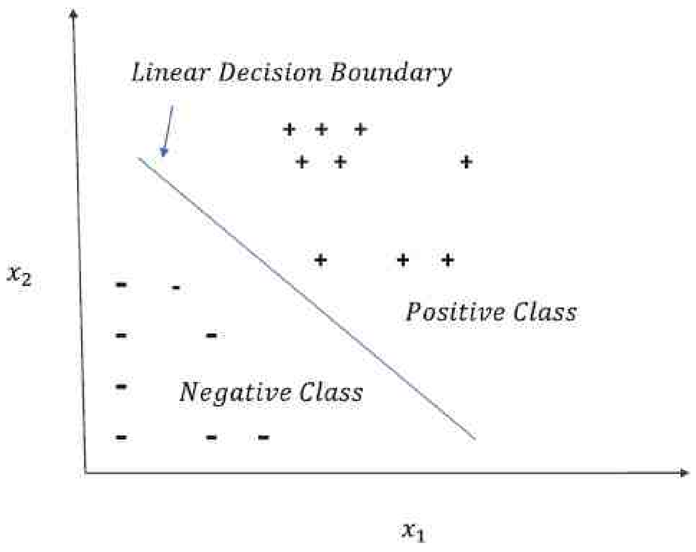


Figure 1-34: Classification by a linear decision boundary

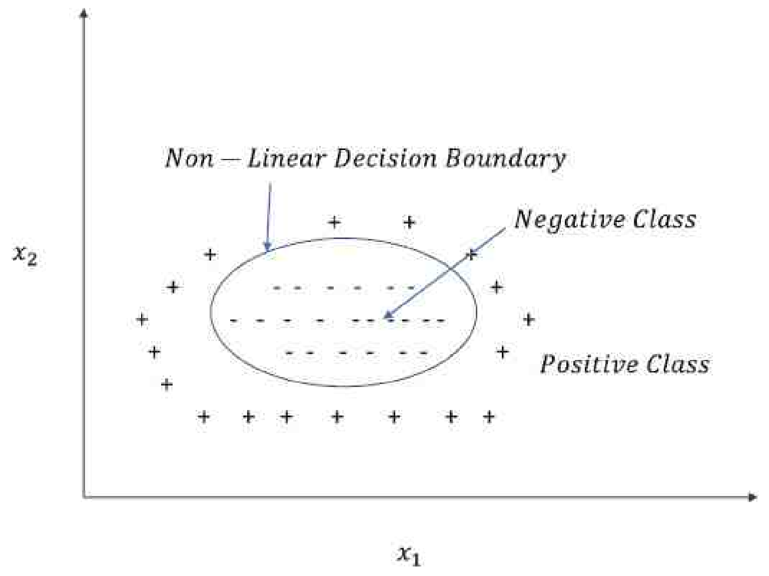


Figure 1-35: Classification by a non-linear decision boundary

I would like to briefly discuss one of the most popular and simple classifiers, logistic regression, so that it becomes clear how the log-loss cost function comes into the picture via maximum likelihood methods in case of classification problems.

Suppose ($x^{(i)}$, $y^{(i)}$) are the labelled data points, where $x^{(i)} \in \mathbb{R}^{n\times l}$ $\forall i \in \{l,2,\cdot\cdot,m\}$ is the input vector, which includes the constant feature value of 1 as a component, and $y^{(i)}$ determines the class. The value of $y^{(i)}$ is set to 1 when the data point corresponds to a customer who has defaulted on his loan and 0 if the customer has not defaulted. The input vector $x^{(i)}$ can be represented in terms of its components as follows:

$$x^{(i)} = \left[1x_0^{(1)} \quad x_1^{(1)}\, x_2^{(1)} x_3^{(1)}\right]$$

Logistic regression in general is a linear classifier and uses a squashing function to convert the linear score into probability. Let $\theta = [\theta_0\ \theta_1\ \theta_2 \ldots \theta_n]$ be the model parameter with each θ_j $\forall j \in \{0,1,2,\ldots,n\}$ representing the model parameter component with respect to the *jth* feature x_j of the input vector x.

The term θ_0 is the bias term. In linear regression, the bias is nothing but the intercept on the output *y* axis. We will look at what this bias term means for logistic regression (and for other linear classifiers) shortly.

The dot product $\theta^T x$ determines whether the given datapoint is likely to be a positive class or a negative class. For the problem at hand, the positive class is the event the customer defaults on his loan repayment and the negative class is the event that the customer doesn't default. The probability that the customer will default given the input and the model is given by the following:

$$P(y=1/x,\theta) = 1/\left(1+exp\left(-\theta^T x\right)\right) = p$$

$$P(y=0/x,\theta) = 1 - 1/\left(1+exp\left(-\theta^T x\right)\right) \;= exp\left(-\theta^T x\right)/\left(1+exp\left(-\theta^T x\right)\right) \;= q$$

Now, let's look at the probability values for different values of $\theta^T x$:

- $\theta^T x = 0$ then the probability of positive class is 1/2.
- When $\theta^T x > 0$ the probability of a positive class is greater than 1/2 and less than 1.
- When $\theta^T x < 0$ the probability of a positive class is less than 1/2 and greater than 0.
- When $\theta^T x$ is sufficiently large and positive, i.e., $\theta^T x \to \infty$, the probability $\to$ 1.
- When $\theta^T x$ is sufficiently large and negative, i.e., $\theta^T x \to -\infty$, the probability $\to$ 0.

The good thing about this probability formulation is that it keeps the values between 0 and 1, which would not have been possible with linear regression. Also, instead of the actual class, it gives continuous probability. Thus, depending on the problem at hand, the cut-off probability thresholds can be defined to determine the class.

This probability model function is called a logistic or sigmoid function. It has smooth, continuous gradients that make the model training mathematically convenient.

If we look carefully we will see that the customer class *y* for each training sample follows a Bernoulli distribution, which we discussed earlier. For every data point, the class $y^{(i)} / x^{(i)} \sim Bernoulli(l, p_i)$. Based on the probability mass function of Bernoulli distribution, we can say the following:

$$P\left(y^{(i)} / x^{(i)}, \theta\right) = \left(1 - p_i\right)^{1-y^{(i)}} \quad \text{where } p_i = 1/\left(1 + exp\left(-\theta^T x\right)\right)$$

Now, how do we define the cost function? We compute the likelihood of the data given the model parameters and then determine the model parameters that maximize the computed likelihood. We define the likelihood by *L*, and it can be represented as

$$L = P(Data / model) = P\left(D^{(1)} D^{(2)} \ldots D^{(m)} / \theta\right)$$

where $D^{(i)}$ represents the *ith* training sample $(x^{(i)}, y^{(i)})$.

Assuming the training samples are independent, given the model, *L* can be factorized as follows:

$$L = P\left(D^{(1)} D^{(2)} \ldots D^{(m)} / \theta\right)$$

$$= P\left(D^{(1)} / \theta\right) P\left(D^{(2)} / \theta\right) \ldots. P\left(D^{(m)} / \theta\right)$$

$$= \prod_{i=1}^{m} P\left(D^{(i)} / \theta\right)$$

We can take the log on both sides to convert the product of probabilities into a sum of the log of the probabilities. Also, the optimization remains the same since the maxima point for both *L* and *logL* would be the same, as log is a monotonically increasing function.

Taking log on both sides, we get the following:

$$logL = \sum_{i=1}^{m} log\, P\left(D^{(i)} / \theta\right)$$

Now, $P\left(D^{(i)} / \theta\right) = P\left(\left(x^{(i)}, y^{(i)}\right) / \theta\right) = P\left(x^{(i)} / \theta\right) P\left(y^{(i)} / x^{(i)}, \theta\right)$.

We are not concerned about the probability of the data point—i.e., $P(x^{(i)}/\theta)$—and assume that all the data points in the training are equally likely for the given model. So, $P(D^{(i)}/\theta) = k\, p(y^{(i)}/x^{(i)}, \theta)$, where *k* is a constant.

Taking the log on both sides, we get the following:

$$log\, P\left(D^{(i)} / \theta\right) = logk + y^{(i)} logp_i + \left(1 - y^{(i)}\right) log\left(1 - p_i\right)$$

Summing over all data points, we get the following:

$$logL = \sum_{i=1}^{m} logk + y^{(i)} logp_i + \left(1 - y^{(i)}\right) log\left(1 - p_i\right)$$

We need to maximize the *logL* to get the model parameter *θ*. Maximizing *logL* is the same as minimizing *–logL*, and so we can take the *–logL* as the cost function for logistic regression and minimize it. Also, we can drop the *logk* sum since it's a constant and the model parameter at the minima would be same irrespective of whether we have the *logk* sum. If we represent the cost function as *C*(*θ*), it can be expressed as seen here:

$$C(\theta)=\sum_{i=1}^{m}-y^{(i)}logp_i-\left(1-y^{(i)}\right)log\left(1-p_i\right)$$

$$\text{where } p_i=1/\left(1+exp\left(-\theta^T x\right)\right)$$

$C(\theta)$ is a convex function in θ, and the reader is encouraged to verify it with the rules learned earlier in the "Calculus" section. $C(\theta)$ can be minimized by common optimization techniques.

Hyperplanes and Linear Classifiers

Linear classifiers in some way or another are related to a hyperplane, so it makes sense to look at that relationship. In a way, learning a linear classifier is about learning about the hyperplane that separates the positive class from the negative class.

A hyperplane in an *n*-dimensional vector space is a plane of dimension (*n*–1) that divides the *n*-dimensional vector space into two regions. One region consists of vectors lying above the hyperplane, and the other region consists of vectors lying below the hyperplane. For a two-dimensional vector space, straight lines act as a hyperplane. Similarly, for a three-dimensional vector space, a two-dimensional plane acts as a hyperplane.

A hyperplane is defined by two major parameters: its perpendicular distance from the origin represented by a bias term *b′* and the orientation of the hyperplane determined by a unit vector *w* perpendicular to the hyperplane surface as shown in Figure 1-36.

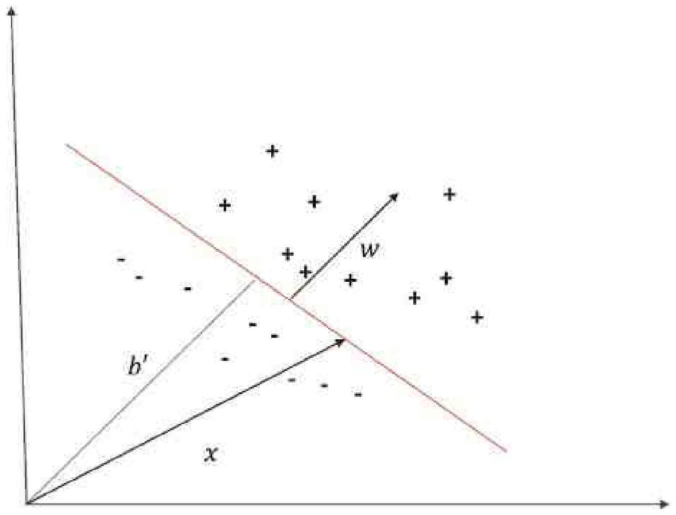


Figure 1-36: Hyperplanes separating the two classes

For a vector $x \in \mathbb{R}^{n\times 1}$ to lie on the hyperplane, the projection of the vector in the direction of *w* should equal the distance of the hyperplane from origin—i.e., $w^Tx = b'$. Thus, any point lying on the hyperplane must satisfy $w^Tx–b' = 0$. Similarly, $w^Tx–b' > 0$ must be satisfied for points lying above the hyperplane and $w^Tx–b' < 0$ for points lying below the hyperplane.

In linear classifiers, we learn to model the hyperplane, or learn the model parameters *w* and *b*. The vector *w* is generally aligned toward the positive class. The Perceptron and linear SVM are linear classifiers. Of course, the ways SVM and Perceptron learn the hyperplane are totally different, and hence they would come up with different hyperplanes, even for the same training data.

Even if we look at logistic regression, we see that it is based on a linear decision boundary. The linear decision boundary is nothing but a hyperplane, and points lying on the hyperplane (i.e., $w^Tx–b' = 0$) are assigned a probability of 0.5 for either of the classes. And again, the way the logistic regression learns the decision boundary is totally different from the way SVM and Perceptron models do.

Unsupervised Learning

Unsupervised machine-learning algorithms aim at findings patterns or internal structures within datasets that contain input data points without labels or targets. *K*-means clustering, the mixture of Gaussians, and so on are methods of unsupervised learning. Even data-reduction techniques like Principal Component Analysis (PCA), Singular Value Decomposition (SVD), auto-encoders, and so forth are unsupervised learning methods.

Optimization Techniques for Machine Learning

Gradient Descent

Gradient descent, along with its several variants, is probably the most widely used optimization technique in machine learning and deep learning. It's an iterative method that starts with a random model parameter and uses the gradient of the cost function with respect to the model parameter to determine the direction in which the model parameter should be updated.

Suppose we have a cost function $C(\theta)$, where θ represents the model parameters. We know the gradient of the cost function with respect to θ gives us the direction of maximum increase of $C(\theta)$ in a linear sense at the value of θ at which the gradient is evaluated. So, to get the direction of maximum decrease of $C(\theta)$ in a linear sense, one should use the negative of the gradient.

The update rule of the model parameter θ at iteration $(t+1)$ is given by

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla C\left(\theta^{(t)}\right)$$

where η represents the learning rate and $\theta^{(t+1)}$ and $\theta^{(t)}$ represent the parameter vector at iteration $(t+1)$ and t respectively.

Once minima is reached through this iterative process, the gradient of the cost function at minima would technically be zero, and hence there would be no further updates to θ. However, because of rounding errors and other limitations of computers, converging to true minima might not be achievable. Thus, we would have to come up with some other logic to stop the iterative process when we believe we have reached minima—or at least close enough to the minima—to stop the iterative process of model training. One of the ways generally used is to check the magnitude of the gradient vector and if it is less than some predefined minute quantity—say, ε—stop the iterative process. The other crude way that can be used is to stop the iterative process of the parameter update after a fixed number of iterations, like 1000.

Learning rate plays a very vital role in the convergence of the gradient descent to the minima point. If the learning rate is large, the convergence might be faster but might lead to severe oscillations around the point of minima. A small learning rate might take a longer time to reach the minima, but the convergence is generally oscillation free.

To see why gradient descent works, let's take a model that has one parameter and a cost function $C(\theta)=(\theta - a)^2 + b$ and see why the method works.

As we can see from Figure 1-37, the gradient (in this case it's just the derivative) at point θ_1 is positive, and thus if we move along the gradient direction the cost function would increase. Instead, at θ_1 if we move along the negative of the gradient the cost reduces. Again, let's take the point θ_2 where the gradient is negative. If we take the gradient direction here for updating θ, the cost function would increase. Taking the negative direction of the gradient would ensure that we move toward the minima. Once we reach the minima at $\theta = a$, the gradient is 0, and hence there would be no further update to θ.

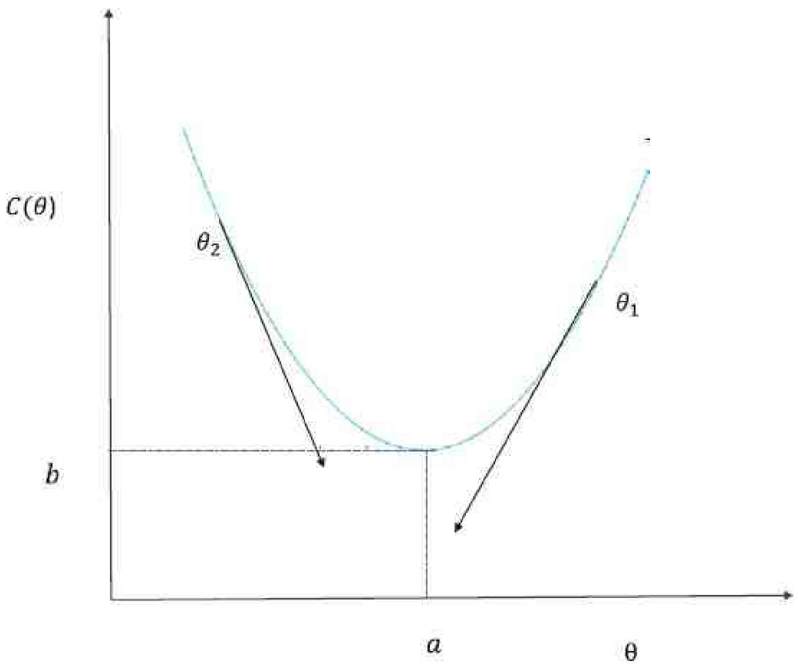


Figure 1-37: Gradient descent intuition with simple cost function in one variable

Gradient Descent for a Multivariate Cost Function

Now that we have some intuition about gradient descent from looking at a cost function based on one parameter, let's look at gradient descent when the cost function is based on multiple parameters.

Let's look at a Taylor Series expansion of a function of multiple variables. Multiple variables can be represented by the vector θ. Let's consider a cost function $C(\theta)$ where $\theta \in \mathbb{R}^{n\times 1}$.

As discussed earlier, the Taylor Series expansion around a point θ can be expressed in matrix notation, as shown here:

$$C(\theta+\Delta\theta)=C(\theta)+\Delta\theta^T\nabla C(\theta)+\frac{1}{2}\Delta\theta^T H(\theta)\Delta\theta+\textit{higher order terms}$$

$$\Delta\theta \rightarrow \textit{Change in } \theta \textit{ vector}$$

$$\nabla C(\theta)\rightarrow \textit{Gradient vector of } C(\theta)$$

$$H(\theta)\rightarrow \textit{Hessian matrix of } C(\theta)$$

Let's suppose at iteration *t* of gradient descent the model parameter is θ and we want update θ to $(\theta + \Delta\theta)$ by making an update of $\Delta\theta$ such that $C(\theta + \Delta\theta)$ is lesser than $C(\theta)$.

If we assume linearity of the function in the neighborhood of θ, then from Taylor Series expansion we get the following:

$$C(\theta+\Delta\theta)=C(\theta)+\Delta\theta^T\nabla C(\theta)$$

We want to choose $\Delta\theta$ in such a way that $C(\theta + \Delta\theta)$ is less than $C(\theta)$.

For all $\Delta\theta$ with the same magnitude, the one that will maximize the dot product $\Delta\theta^T \nabla C(\theta)$ should have a direction the same as that of $\nabla C(\theta)$. But that would give us the maximum possible $\Delta\theta^T \nabla C(\theta)$. Hence, to get the minimum value of the dot product $\Delta\theta^T \nabla C(\theta)$, the direction of $\Delta\theta$ should be the exact opposite of that of $\nabla C(\theta)$.V In other words, $\Delta\theta$ should be proportional to the negative of the gradient vector $\nabla C(\theta)$:

$$\Delta\theta \propto -\nabla C(\theta)$$

$=> \Delta\theta = -\eta \nabla C(\theta)$, where η is the learning rate

$=> \theta + \Delta\theta = \theta - \eta \nabla C(\theta)$

$=> \theta^{(t+1)} = \theta^{(t)} - \eta \nabla C\left(\theta^{(t)}\right)$

which is the famous equation for gradient descent. To visualize how the gradient descent proceeds to the minima, we need to have some basic understanding of contour plots and contour lines.

Contour Plot and Contour Lines

Let us consider a function $C(\theta)$ where $\theta \in \mathbb{R}^{n\times 1}$. A contour line is a line/curve in the vector space of θ that connects points that have the same value of the function $C(\theta)$. For each unique value of $C(\theta)$ we would have separate contour lines.

Let's plot the contour lines for a function $C(\theta) = \theta^T A\theta$, where $\theta = [\theta_1 \theta_2]^T \in \mathbb{R}^{2\times 1}$ and

$$A = \begin{bmatrix} 7 & 2 \\ 2 & 5 \end{bmatrix}$$

Expanding the expression for $C(\theta)$ we get

$$C(\theta_1, \theta_2) = 7\theta_1^2 + 5\theta_2^2 + 4\theta_1\theta_2$$

The contour plot for the cost function is depicted in Figure 1-38.

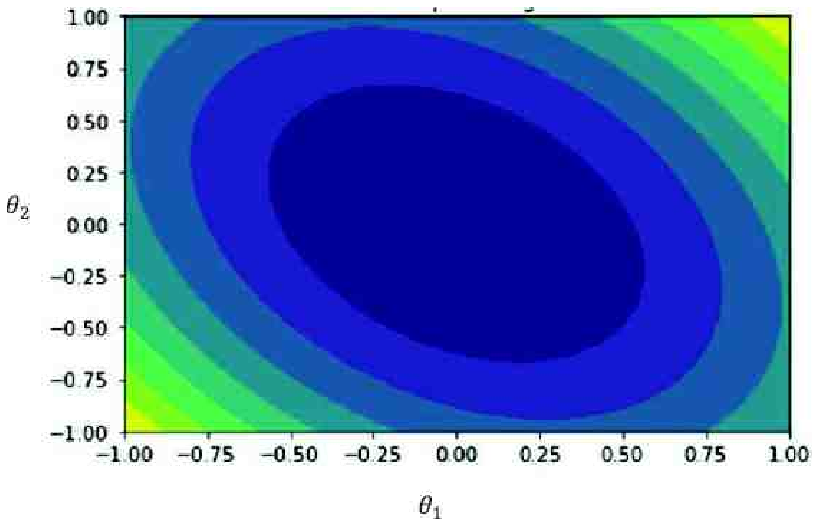


Figure 1-38: Contour plots

Each of the ellipses are contour lines specific to a fixed value of the function $C(\theta_1,\theta_2)$. If $C(\theta_1,\theta_2) = a$, where a is a constant, then the equation

$a = 7\theta_1^2 + 5\theta_2^2 + 4\theta_1\theta_2$ represents an ellipse.

For different values of constant a we get different ellipses, as depicted in Figure 1-38. All points on a specific contour line have the same value of the function.

Now that we know what a contour plot is, let's look at gradient descent progression in a contour plot for the cost function $C(\theta)$, where $\theta \in \mathbb{R}^{2\times 1}$. The gradient descent steps have been illustrated in Figure 1-39.

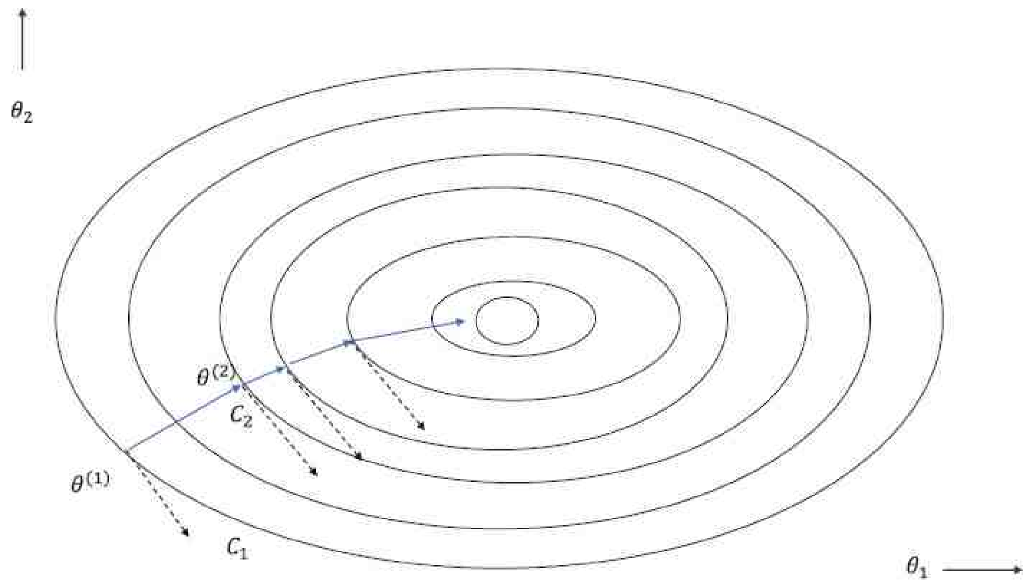


Figure 1-39: Gradient descent for a two-variable cost function

Let's take the largest ellipse corresponding to cost C_1 and assume our current θ is at the point $\theta^{(1)}$ in C_1.

Assuming the linearity of the $C(\theta)$ about θ, the change in cost function $C(\theta)$ can be presented as seen here:

$$\Delta C(\theta) = \Delta\theta^T \nabla C(\theta)$$

If we take a small change in cost between two points very close to each other in the same contour line, then $\Delta C(\theta) = 0$, since all points on the same contour line have the same fixed value. Also, it should be noted that when we take two points very close to each other on the same contour line, the $\Delta\theta$ represents the tangent to the contour line represented by tangential arrows to the contour line. Please do not confuse this $\Delta\theta$ to be the $\Delta\theta$ update to the parameter in gradient descent.

$\Delta C(\theta) = 0 \rightarrow \Delta\theta^T \nabla C(\theta) = 0$, which basically means that the gradient is perpendicular to the tangent at the point $\theta^{(1)}$ in the contour line C_1. The gradient would have pointed outward, whereas the negative of the gradient points inward, as depicted by the arrow perpendicular to the tangent. Based on the learning rate, it will reach a point $\theta^{(2)}$ in a different contour line represented by C_2, whose cost function value would be less than that of C_1. Again, the gradient would be evaluated at $\theta^{(2)}$, and the same process would be repeated for several iterations until it reached the minima point, where the gradient would drop to 0 technically, after which there would be no more updates to θ.

Steepest Descent

Steepest descent is a form of gradient descent where the learning rate is not constant but rather is computed at every iteration to ensure that the parameter update through gradient descent takes the cost function to minima with respect to the learning rate. In other words, the learning rate at every iteration in steepest descent is optimized to ensure that the movement in the direction of the gradient is utilized to the maximum extent.

Let us take our usual cost function $C(\theta)$ and look at successive iterations t and (t+l). As with gradient descent, we have the parameter update rule as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla C\left(\theta^{(t)}\right)$$

So, the cost function at iteration (t+l) can be expressed as

$$C\left(\theta^{(t+1)}\right) = C\left(\theta^{(t)} - \eta \nabla C\left(\theta^{(t)}\right)\right)$$

To minimize the cost function at iteration (t+1) with respect to the learning rate, see the following:

$$\frac{\partial C\left(\theta^{(t+1)}\right)}{\partial \eta} = 0$$

$$=> \nabla C\left(\theta^{(t+1)}\right) \frac{\partial\left[C(\theta^{(t)} - \eta \nabla C\left(\theta^{(t)}\right)\right]}{\partial \eta} = 0$$

$$=> -\nabla C\left(\theta^{(t+1)}\right)^T \nabla C\left(\theta^{(t)}\right) = 0$$

$$=> \nabla C\left(\theta^{(t+1)}\right)^T \nabla C\left(\theta^{(t)}\right) = 0$$

So, for steepest descent, the dot product of the gradients at (t+1) and t is 0, which implies that the gradient vector at every iteration should be perpendicular to the gradient vector at its previous iteration.

Stochastic Gradient Descent

Both steepest descent and gradient descent are full-batch models; i.e., the gradients are computed based on the whole training dataset. So, if the dataset is huge, the gradient computation becomes expensive and the memory requirement increases. Also, if the dataset has huge redundancy, then computing the gradient on the full dataset is not useful since similar gradients can be computed by using much smaller batches called mini batches. The most popular method to overcome the preceding problems is to use an optimization technique called stochastic gradient descent.

Stochastic gradient descent is a technique for minimizing a cost function based on the gradient descent method where the gradient at each step is not based on the entire dataset but rather on single data points.

Let $C(\theta)$ be the cost function based on m training samples. The gradient descent at each step is not based on $C(\theta)$ but rather on $C^{(i)}(\theta)$, which is the cost function based on the *ith* training sample. So, if we must plot the gradient vectors at each iteration against the overall cost function, the contour lines would not be perpendicular to the tangents since they are based on gradients of $C^{(i)}(\theta)$ and not on the overall $C(\theta)$.

The cost functions $C^{(i)}(\theta)$ are used for gradient computation at each iteration, and the model parameter vector θ is updated by the standard gradient descent in each iteration until we have made a pass over the entire training dataset. We can perform several such passes over the entire dataset until a reasonable convergence is obtained.

Since the gradients at each iteration are not based on the entire training dataset but rather on single training samples, they are generally very noisy and may change direction rapidly. This may lead to oscillations near the minima of the cost function, and hence the learning rate should be less while converging to the minima so that the update to the parameter vector is as small as possible. The gradients are cheaper and faster to compute, and so the gradient descent tends to converge faster.

One thing that is important in stochastic gradient descent is that the training samples should be as random as possible. This will ensure that a stochastic gradient descent over a period of a few training samples provides a similar update to the model parameter as that resulting from an actual gradient descent, since the random samples are more likely to represent the total training dataset. If the samples at each iteration of stochastic gradient descent are biased, they don't represent the actual dataset, and hence the update to the model parameter might be in a direction that would result in it taking a long time for the stochastic gradient descent to converge.

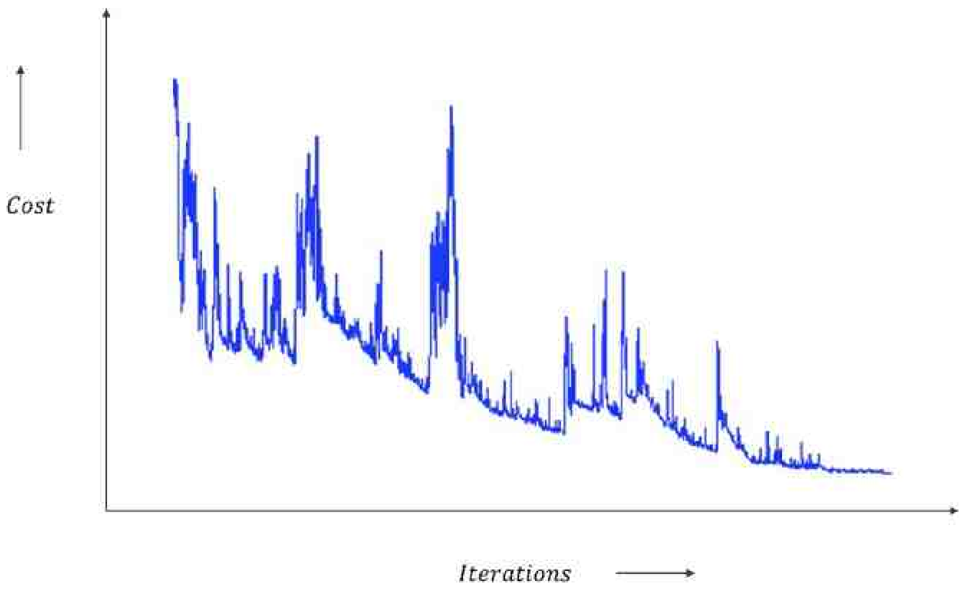


Figure 1-40: Fluctuation in the total cost function value over iterations in stochastic gradient descent

As illustrated in Figure 1-41, the gradients at each step for stochastic gradient descent are not perpendicular to the tangents at the contour lines. However, they would be perpendicular to the tangents to the contour lines for individual training samples had we plotted them. Also, the associated cost reduction over iterations is noisy because of the fluctuating gradients.

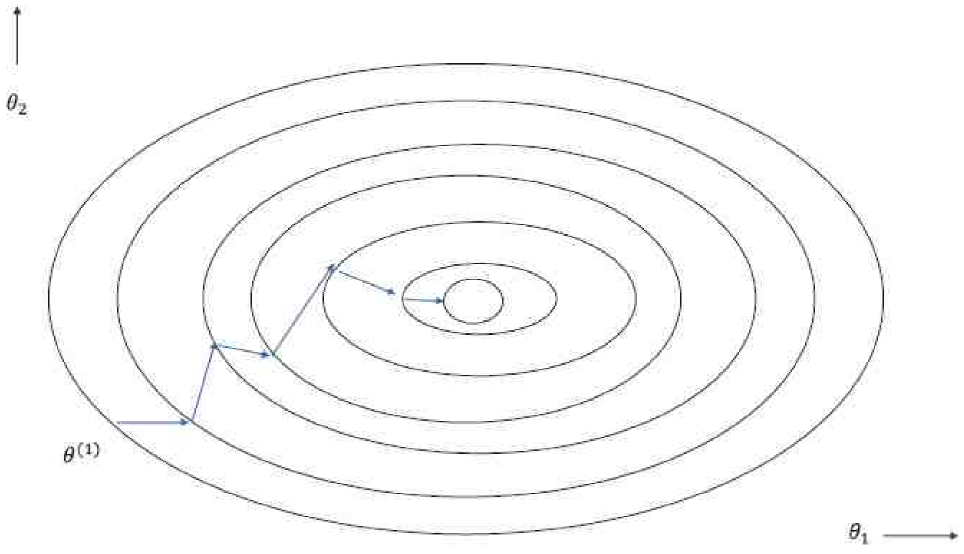


Figure 1-41: Stochastic gradient descent parameter update

The gradient computations become very cheap when we use single training data points. Also, convergence is quite fast, but it does come with its own set of disadvantages, as follows:

- Since the estimated gradients at each iteration are not based on the total cost function but rather on the cost function associated with single data points, the gradients are very noisy. This leads to convergence problems at the minima and may lead to oscillations.
- The tuning of the learning rate becomes important since a high learning rate might lead to oscillations while converging to the minima. This is because the gradients are very noisy, and hence if the gradient estimates at the convergence are not near to zero a high learning rate will take the update well past the minima point and the process can repeat on either side of the minima.
- Since the gradients are noisy, the model parameter values after each iteration are also very noisy, and thus heuristics need to be added to the stochastic gradient descent to determine which value of model parameter to take. This also brings about another question: when to stop the training.

A compromise between full-batch model and stochastic gradient descent is a mini-batch approach wherein the gradient is neither based on the full training dataset nor on the single data points. Rather, it uses a mini batch of training data points to compute the cost function. Most of the deep-learning algorithms use a mini-batch approach for stochastic gradient descent. The gradients are less noisy and at the same time don't cause many memory constraints because the mini-batch sizes are moderate.

We will discuss mini-batches in more detail in Chapter 2.

Newton's Method

Before we start Newton's method for optimizing a cost function for its minima, let's look at the limitations of gradient-descent techniques.

Gradient-descent methods rely on the linearity of the cost function between successive iterations; i.e., the parameter value at iteration (t+1) can be reached from the parameter value at time t by following the gradient, since the path of the cost function $C(\theta)$ from t to (t+1) is linear or can be joined by a straight line. This is a very simplified assumption and would not yield good directions for gradient descent if the cost function is highly non-linear or has curvature. To get a better idea of this, let's look at the plot of a cost function with a single variable for three different cases.

Linear Curve

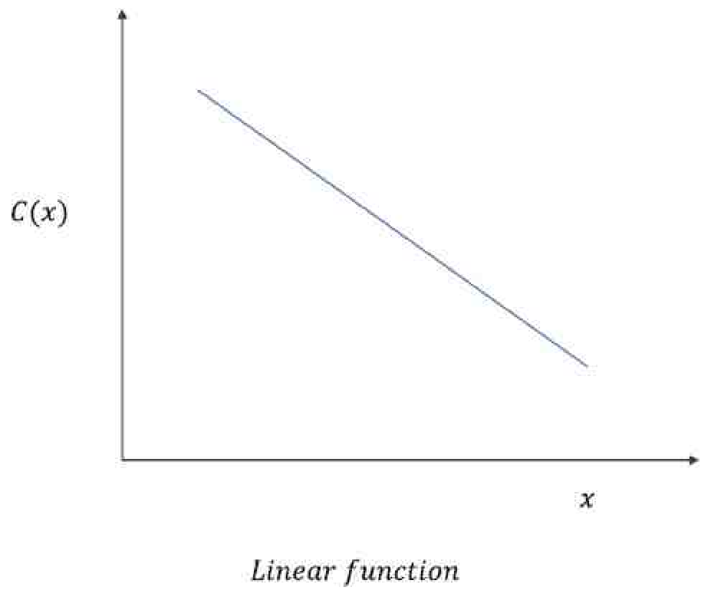


Figure 1-42: Linear cost function

Negative Curvature

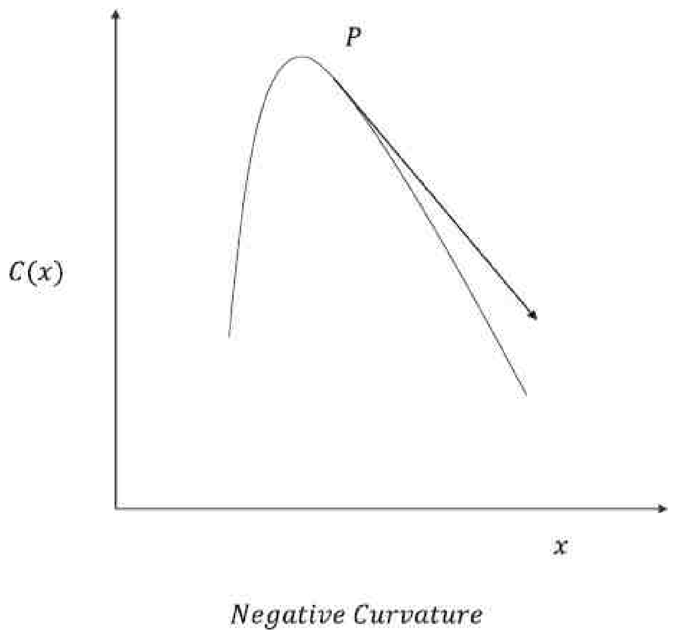


Figure 1-43: Cost function with negative curvature at point P

Positive Curvature

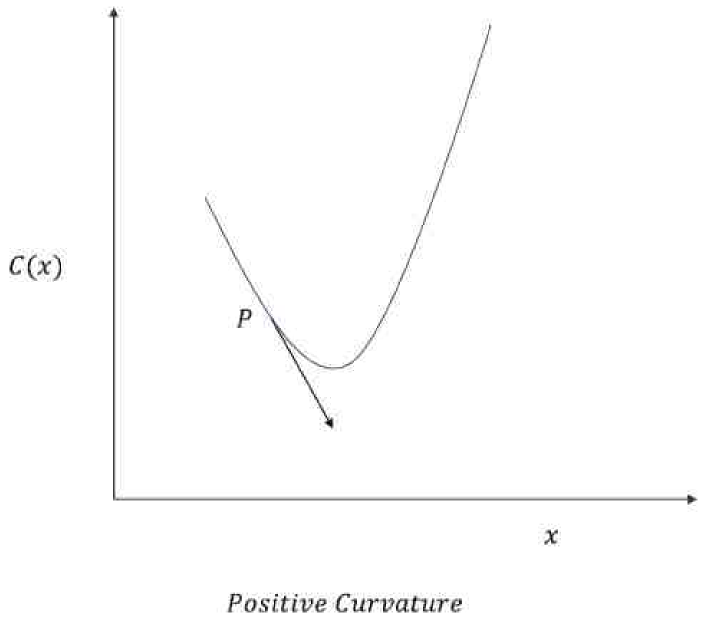


Figure 1-44: Cost function with positive curvature at point P

For the linear cost function, as shown in Figure 1-42, the negative of the gradient would give us the best direction for reaching the minima since the function is linear and doesn't have any curvature. For both the negative and positive curvature cost functions, shown in Figures 1-43 and 1-44 respectively, the derivative would not give us a good direction for minima, and so to take care of the curvature we would need the Hessian along with the derivative. Hessians, as we have seen, are nothing but a matrix of second derivatives. They contain information about the curvature and thus would give a better direction for a parameter update as compared to a simple gradient descent.

Gradient-descent methods are first-order approximation methods for optimization whereas Newton's methods are second-order methods for optimization since they use the Hessian along with the gradient to take care of curvatures in the cost function.

Let's take our usual cost function $C(\theta)$, where $\theta \in \mathbb{R}^{n\times 1}$ is an n-dimensional model parameter vector. We can approximate the cost function $C(\theta)$ in the neighborhood of θ by its second-order Taylor series expansion, as shown here:

$$C(\theta+\Delta\theta)=C(\theta)+\Delta\theta^T\nabla C(\theta)+\frac{1}{2}\Delta\theta^T H(\theta)\Delta\theta$$

$\nabla C(\theta)$ is the gradient and $H(\theta)$ is the Hessian of the cost function $C(\theta)$.

Now, if θ is the value of the model parameter vector at iteration *t*, and $(\theta + \Delta\theta)$ is the value of the model parameter at iteration $(t+1)$, then

$$C\left(\theta^{(t+1)}\right) = C\left(\theta^{(t)}\right) + \Delta\theta^T \nabla C\left(\theta^{(t)}\right) + \frac{1}{2}\Delta\theta^T H\left(\theta^{(t)}\right)\Delta\theta$$

$$\text{where } \Delta\theta = \theta^{(t+1)} - \theta^{(t)}.$$

Taking the gradient with respect to $\theta^{(t+1)}$ we have

$$\nabla C\left(\theta^{(t+1)}\right) = \nabla C\left(\theta^{(t)}\right) + H\left(\theta^{(t)}\right)\Delta\theta$$

Setting the gradient $\nabla C(\theta^{(t+1)})$ to 0 we get

$$\nabla C\left(\theta^{(t)}\right) + H\left(\theta^{(t)}\right)\Delta\theta = 0$$

$$=> \Delta\theta = -H\left(\theta^{(t)}\right)^{-1}\nabla C\left(\theta^{(t)}\right)$$

So, the parameter update for Newton's method is as follows:

$$=> \theta^{(t+1)} = \theta^{(t)} - H\left(\theta^{(t)}\right)^{-1}\nabla C\left(\theta^{(t)}\right)$$

We don't have a learning rate for Newton's method, but one may choose to use a learning rate, much like with gradient descent. Since the directions for non-linear cost functions are better with Newton's method, the iterations to converge to the minima would be fewer as compared to gradient descent. One thing to note is that if the cost function that we are trying to optimize is a quadratic cost function, such as the one in linear regression, then Newton's method would technically converge to the minima in one step.

However, computing the Hessian matrix and its inverse is computationally expensive or intractable at times, especially when the number of input features is large. Also, at times there might be functions for which the Hessian is not even properly defined. So, for large machine-learning and deep-learning applications, gradient descent—especially Stochastic gradient descent—techniques with mini batches are used since they are relatively less computationally intensive and scale up well when the data size is large.

Constrained Optimization Problem

In a constrained optimization problem, along with the cost function that we need to optimize, we have a set of constraints that we need to adhere to. The constraints might be equations or inequalities.

Whenever we want to minimize a function that is subject to an equality constraint, we use the Lagrange formulation. Let's say we must minimize $f(\theta)$ subject to $g(\theta) = 0$ where $\theta \in \mathbb{R}^{n\times 1}$. For such a constrained optimization problem, we need to minimize a function $L(\theta, \lambda) = f(\theta) + \lambda g(\theta)$. Taking the gradient of *L*, which is called the Lagrangian, with respect to the combined vector θ, λ, and setting it to 0 would give us the required θ that minimizes $f(\theta)$ and adheres to the constraint. λ is called the Lagrange multiplier. When there are several constraints, we need to add all such constraints, using a separate Lagrange multiplier for each constraint. Let's say we want to minimize $f(\theta)$ subject to *m* constraints $g_i(\theta) = 0$ $\forall i \in \{1,2,3,\ldots m\}$; the Lagrangian can be expressed as follows:

$$L(\theta,\lambda) = f(\theta) + \sum_{i=1}^{m}\lambda_i g_i(\theta)$$

$$\text{where } \lambda = [\lambda_1 \lambda_2 \quad .. \lambda_m]^T$$

To minimize the function, the gradient of $L(\theta, \lambda)$ with respect to both θ and λ vectors should be a zero vector; i.e.,

$$\nabla_{\theta}(\theta,\lambda)=0$$

$$\nabla_{\lambda}(\theta,\lambda)=0$$

The preceding method can't be directly used for constraints with inequality. In such cases, a more generalized approach called the Karush Kahn Tucker method can be used.

Let $C(\theta)$ be the cost function that we wish to minimize, where $\theta \in \mathbb{R}^{n\times 1}$. Also, let there be k number of constraint on θ such that

$$f_1(\theta)=a_1$$

$$f_2(\theta)=a_2$$

$$f_3(\theta)\le a_3$$

$$f_4(\theta)\ge a_4$$

…

…

$$f_k(\theta)=a_k$$

This becomes a constrained optimization problem since there are constraints that θ should adhere to. Every inequality can be transformed into a standard form where a certain function is less than or less than equal to zero. For example:

$$f_4(\theta)\ge a_4 => -f_4(\theta)\le -a_4 => -f_4(\theta)+a_4\le 0$$

Let each such constraint strictly less than, or less than equal to, zero be represented by $g_i(\theta)$. Also, let there be some strict equality equations $e_j(\theta)$. Such minimization problems are solved through the Karush Kuhn Tucker version of Lagrangian formulation.

Instead of minimizing $C(\theta)$, we need to minimize a cost function $L(\theta, \alpha, \beta)$ as follows:

$$L(\theta,\alpha,\beta)=C(\theta)+\sum_{i=1}^{k_1}\alpha_i g_i(\theta)+\sum_{j=1}^{k_2}\beta_j e_j(\theta)$$

The scalers $\alpha_i\ \forall\ i \in \{l,2,3,\cdot\cdot k_1\}$ and $\beta_j\ \forall\ j \in \{l,2,3,\cdot\cdot k_2\}$ are called the Lagrangian multipliers, and there would be k of them corresponding to k constraints. So, we have converted a constrained minimization problem into an unconstrained minimization problem.

To solve the problem, the Karush Kuhn Tucker conditions should be met at the minima point as follows:

- The gradient of $L(\theta, \alpha, \beta)$ with respect to θ should be the zero vector; i.e.,

$$\nabla_{\theta}(\theta,\alpha,\beta)=0$$

$$=>\nabla_{\theta}C(\theta)+\sum_{i=1}^{k_1}\alpha_i\nabla_{\theta}g_i(\theta)+\sum_{j=1}^{k_2}\beta_j\nabla_{\theta}e_j(\theta)=0$$

- The gradient of the $L(\theta, \alpha, \beta)$ with respect to β, which is the Lagrange multiplier vector corresponding to the equality conditions, should be zero:

$$\nabla_\beta(\theta,\alpha,\beta)=0$$

$$=>\nabla_\beta C(\theta)+\sum_{i=1}^{k_1}\alpha_i\nabla_\beta g_i(\theta)+\sum_{j=1}^{k_2}\beta_j\nabla_\beta e_j(\theta)=0$$

- The inequality conditions should become equality conditions at the minima point. Also, the inequality Lagrange multipliers should be non-negative:

$$\alpha_i g_i(\theta)=0 \ and\, \alpha_i \geq 0 \quad \forall i \in \{1,2,\ldots,k_1\}$$

Solving for the preceding conditions would provide the minima to the constrained optimization problem.

A Few Important Topics in Machine Learning

In this section, we will discuss a few important topics that are very much relevant to machine learning. Their underlying mathematics is very rich.

Dimensionality Reduction Methods

Principal component analysis and singular value decomposition are the most commonly used dimensionality-reduction techniques in the machine-learning domain. We will discuss these techniques to some extent here. Please note that these data-reduction techniques are based on linear correlation and don't capture non-linear correlation such as co-skewness, co-Kurtosis, and so on. We will talk about a few dimensionality-reduction techniques that are based on artificial neural networks, such as auto-encoders, in the latter part of the book.

Principal Component Analysis

Principal component analysis is a dimensionality-reduction technique that ideally should have been discussed in the "Linear Algebra" section. However, to make its mathematics much easier to grasp, I intentionally kept it for after the constrained optimization problem. Let's look at the two-dimensional data plot in Figure 1-45. As we can see, the maximum variance of the data is neither along the *x* direction nor along the *y* direction, but somewhat in a direction in between. So, had we projected the data in a direction where the variance is at maximum, it would have covered most of the variability in the data. Also, the rest of the variance could have been ignored as noise.

The data along the *x* direction and the *y* direction are highly correlated (see Figure 1-45). The covariance matrix would provide the required information about the data to reduce the redundancy. Instead of looking at the *x* and *y* directions we can look at the a_1 direction, which has the maximum variance.

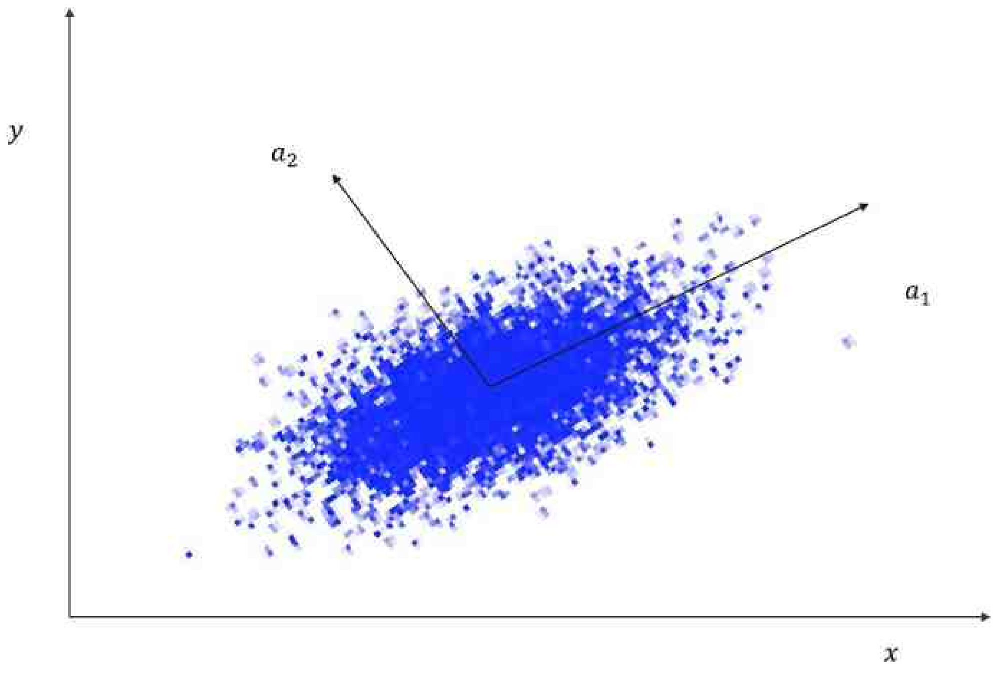


Figure 1-45: Correlated 2-D data. a_1 and a_2 are the directions along which the data is uncorrelated and are the principal components.

Now, let's get into some math and assume that we don't have the plot and we only have the data for *m* number of samples $x^{(i)} \in \mathbb{R}^{n \times 1}$.

We want to find out independent directions in the n-dimensional plane in the order of decreasing variances. By independent directions I mean the covariance between those directions should be 0. Let a_1 be the unit vector along which the variance of the data is maximum. We first subtract the mean of the data vector to center the data on the origin. Let μ be the mean vector of the data vector x; i.e., $E[x] = \mu$.

A component of the $(x - \mu)$ vector in the direction of a_1 is the projection of $(x - \mu)$ on a_1; let it be denoted by z_1:

$$z_1 = a_1^T(x - \mu)$$

$var(z_1) = var[a_1^T(x - \mu)] = a_l^T cov(x) a_l$, where *var* denotes variance and *cov*(*x*) denotes the covariance matrix.

For given data points the variance is a function of a_1. So, we would have to maximize the variance with respect to a_1 given that a_1 is a unit vector:

$$a_1^T a_1 = 1 => a_1^T a_1 - 1 = 0$$

So, we can express the function to be maximized as $L(a_1, \lambda) = a_1^T cov(x) a_1 - \lambda(a_1^T a_1 - 1)$, where λ is a Lagrangian multiplier.

For maxima, setting the gradient of L with respect to $a_1$to 0, we get

$$\nabla L = 2cov(x)a_1 - 2\lambda a_1 = 0 => cov(x)a_1 = \lambda a_1$$

We can see something come up that we studied earlier. The a_1 vector is nothing but an Eigen vector of the covariance matrix, and λ is the corresponding Eigen value.

Now, substituting this into the variance expression along a_1, we get

$$var(z_1) = a_1^T cov(x) a_1 = a_1^T \lambda a_1 = \lambda a_1^T a_1 = \lambda$$

Since the expression for variance along a_1 is the Eigen value itself, the Eigen vector corresponding to the highest Eigen value gives us the first principal component, or the direction along which the variance in data is maximum.

Now, let's get the second principal component, or the direction along which the variance is maximum right after a_1.

Let the direction of the second principal component be given by the unit vector a_2. Since we are looking for orthogonal components, the direction of a_2 should be perpendicular to a_1.

A projection of the data along a_2 can be expressed by the variable $z_2 = a_2^T(x - \mu)$.

Hence, the variance of the data along a_2 is $Var(z_2) = Var[a_2^T(x - \mu)] = a_2^T cov(x) a_2$.

We would have to maximize $Var(z_2)$ subject to the constraints $a_2^T a_2 = 1$ since a_2 is a unit vector and $a_2^T a_1 = 0$ since a_2 should be orthogonal to a_1.

We need to maximize the following function $L(a_2, \alpha, \beta)$ with respect to the parameters a_2, α, β:

$$L(a_2, \alpha, \beta) = a_2^T cov(x) a_2 - \alpha(a_2^T a_2 - 1) - \beta(a_2^T a_1)$$

By taking a gradient with respect to a_2 and setting it to zero vector we get

$$\nabla L = 2cov(x)a_2 - 2\alpha a_2 - \beta a_1 = 0$$

By taking the dot product of the gradient ∇L with vector a_1 we get

$$2a_1^T cov(x)a_2 - 2\alpha a_1^T a_2 - \beta a_1^T a_1 = 0$$

$a_1^T cov(x) a_2$ is a scalar and can be written as $a_2^T cov(x) a_1$.

On simplification, $a_2^T cov(x) a_1 = a_2^T \lambda a_1 = \lambda a_2^T a_1 = 0$. Also, the term $2\alpha a_l^T a_2$ is equal to 0, which leaves $\beta a_1^T a_1 = 0$. Since $a_1^T a_1 = 1$, β must be equal to 0.

Substituting $\beta = 0$ in the expression for $\nabla L = 2cov(x)a_2 - 2\alpha a_2 - \beta a_1 = 0$, we have the following:

$$2cov(x)a_2 - 2\alpha a_2 = 0 \; i.e. \; cov(x)a_2 = \alpha a_2$$

Hence, the second principal component is also an Eigen vector of the covariance matrix, and the Eigen value α must be the second-largest Eigen value just after λ. In this way, we would get n Eigen vectors from the covariance matrix cov(x) $\in \mathbb{R}^{n\times 1}$, and the variance of the data along each of those Eigen value directions (or principal components) would be represented by the Eigen values. One thing to note is that the covariance matrix is always symmetrical and thus the Eigen vectors would always be orthogonal to each other and so would give independent directions.

The covariance matrix is always positive semi-definite.

This is true because the Eigen values of the covariance matrix represent variances, and variance can't be negative. If $cov(x)$ is positive definite, i.e., $a^T cov(x) a > 0$, then all the Eigen values of covariance matrices are positive.

Figure 1-46 illustrates a principal component analysis transformation of the data. As we can see, PCA has centered the data and got rid of correlation among the PCA transformed variables.

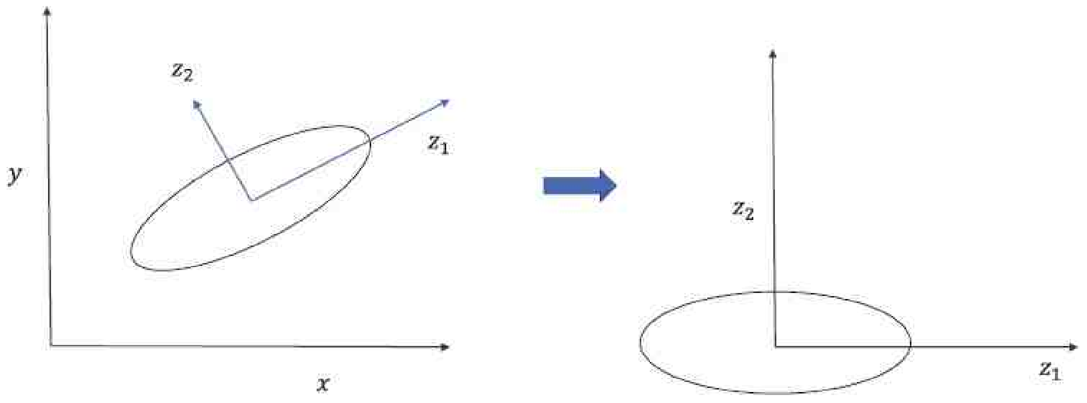


Figure 1-46: Principal component analysis centers the data and then projects the data into axes along which variance is maximum. The data along z_2 can be ignored if the variance along it is negligible.

When Will PCA Be Useful in Data Reduction?

When there is high correlation between the different dimensions of the input, there would only be a few independent directions along which the variance of data would be high, and along other directions the variance would be insignificant. With PCA, one can keep the data components in the few directions in which variance is high and make a significant contribution to the overall variance, ignoring the rest of the data.

How Do You Know How Much Variance Is Retained by the Selected Principal Components?

If the z vector presents the transformed components for input vector x, then the $cov(z)$ would be a diagonal matrix containing the Eigen values of the $cov(x)$ matrix as its diagonal entries.

$$cov(z) = \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \lambda_2 & \vdots \\ 0 & \cdots & \lambda_n \end{bmatrix}$$

Also, let's suppose the Eigen values are ordered like $\lambda_1 > \lambda_2 > \lambda_3 .. > \lambda_n$.

Let's suppose we choose to keep only the first *k* principal components; the proportion of variance in the data captured is as follows:

$$\frac{\lambda_1 + \lambda_2 + \lambda_3 + \ldots + \lambda_k}{\lambda_1 + \lambda_2 + \lambda_3 + \ldots + \lambda_k + \ldots + \lambda_n}$$

Singular Value Decomposition

Singular value decomposition is a dimensionality-reduction technique that factorizes a matrix $A \in \mathbb{R}^{m \times n}$ into a product of three matrices, as $A = USV^T$ where

$U \in \mathbb{R}^{m \times m}$ and is composed of all the Eigen vectors of the matrix AA^T

$V \in \mathbb{R}^{n \times n}$ and is composed of all the Eigen vectors of the matrix A^TA

$S \in \mathbb{R}^{m \times n}$ and is composed of *k* square root of the Eigen vectors of both A^TA and AA^T, where *k* is the rank of matrix *A*.

The column vectors of *U* are all orthogonal to each other and hence form an orthogonal basis. Similarly, the column vectors of *V* also form an orthogonal basis:

$$U = \left[u_1 u_2 \ \ldots \ u_m\right]$$

where $u_i \in \mathbb{R}^{m \times 1}$ are the column vectors of *U*.

$$V = \left[v_1 v_2 \ \cdots \ v_m\right]$$

where $v_i \in \mathbb{R}^{n \times 1}$ are the column vectors of *V*.

$$S = \begin{bmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \sigma_2 & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

Depending on the rank of *A*, there would be $\sigma_1, \sigma_2 \ldots \ldots \ldots \sigma_k$ diagonal entries corresponding to the rank *k* of matrix *A*:

$$A = \sigma_1 u_1 v_1^{\ T} + \sigma_2 u_2 v_2^{\ T} + \ldots . + \sigma_k u_k v_k^{\ T}$$

The $\sigma i \ \forall \ \in \{1,2,3,\cdot\cdot k\}$, also called singular values, are the square root of the Eigen values of both A^TA and AA^T, and hence they are measures of variance in the data. Each of the $\sigma_i u_i v_i^{\ T} \ \ \forall i \in \{1,2,3,..k\}$ is a rank-one matrix. We can only keep the rank-one matrices for which the singular values are significant and explain a considerable proportion of the variance in the data.

If one takes only the first *p* rank-one matrices corresponding to the first *p* singular values of largest magnitude, then the variance retained in the data is given by the following:

$$\frac{\sigma_1^{\ 2} + \sigma_2^{\ 2} + \sigma_3^{\ 2} + \ldots + \sigma_p^{\ 2}}{\sigma_1^{\ 2} + \sigma_2^{\ 2} + \sigma_3^{\ 2} + \ldots + \sigma_p^{\ 2} + \ldots + \sigma_k^{\ 2}}$$

Images can be compressed using singular value decomposition. Similarly, singular value decomposition is used in collaborative filtering to decompose a user-rating matrix to two matrices containing user vectors and item vectors. The singular value decomposition of a matrix is given by USV^T. The user-ratings matrix *R* can be decomposed as follows:

$$R = USV^T = US^{\frac{1}{2}}S^{\frac{1}{2}}V^T = U'V'^T$$

where *U′* is the user-vector matrix and is equal to $US^{\frac{1}{2}}$ and *V′* is the items-vector matrix where $V' = S^{\frac{1}{2}}V^T$.

Regularization

The process of building a machine-learning model involves deriving parameters that fit the training data. If the model is simple, then the model lacks sensitivity to the variation in data and suffers from high bias. However, if the model is too complex, it tries to model for as much variation as possible and in the process models for random noise in the training data. This removes the bias produced by simple models but introduces high variance; i.e., the model is sensitive to very small changes in the input. High variance for a model is not a good thing, especially if the noise in the data is considerable. In such cases, the model in the pursuit of performing too well on the training data performs poorly on the test dataset since the model loses its capability to generalize well with the new data. This problem of models' suffering from high variance is called *overfitting*.

As we can see in Figure 1-47, we have three models fit to the data. The one parallel to the horizontal is suffering from high bias while the curvy one is suffering from high variance. The straight line in between at around 45 degrees to the horizontal has neither high variance nor high bias.

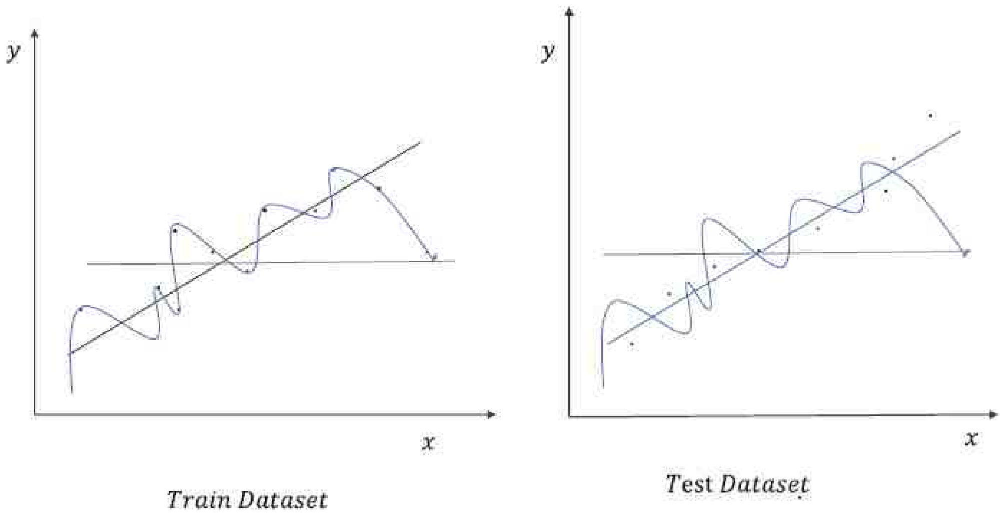


Figure 1-47: Illustration of models with high variance and high bias

The model with high variance does well on the train data but fails to do well on the test dataset, even when the dataset's nature hasn't changed much. The model represented in blue color may not fit the training perfectly, but it does better on the test data since the model doesn't suffer from high variance. The trick is to have a model that doesn't suffer from high bias and at the same time isn't so complex that it models for random noise as well.

Models with high variance typically have model parameters with a large magnitude since the sensitivity of the models to small changes in data is high. To overcome the problem of overfitting resulting from high model variance, a popular technique called regularization is widely used.

To put things into perspective, let's look at the linear regression cost function that we looked at earlier:

$$C(\theta)=\|X\theta - Y\|_2^2$$

$$=(X\theta - Y)^T(X\theta - Y)$$

As discussed earlier, models with high variance have model parameters with a large magnitude. We can put an extra component into the cost function $C(\theta)$ that penalizes the overall cost function in case the magnitude of the model parameter vector is high.

So, we can have a new cost function, $L(\theta)=\|X\theta - Y\|_2^2+\lambda\|\theta\|_2^2$, where $||\theta||_2^2$ is the square of the l^2 norm of the model parameter vector. The optimization problem becomes

$$\theta^* = \underbrace{Arg\ Min}_{\theta}\ L(\theta)=\|X\theta - Y\|_2^2+\lambda\|\theta\|_2^2$$

Taking the gradient ∇L with repect to θ and setting it to 0 gives us $\theta^* = (X^TX+\lambda I)^{-1}X^TY$

Now, as we can see because of the $||\theta||_2^2$ term in the cost function, the model parameter's magnitude can't be too large since it would penalize the overall cost function. λ determines the weight of the regularization term. A higher value for λ would result in smaller values of $||\theta||_2^2$, thus making the model simpler and prone to high bias or underfitting. In general, even smaller values of λ go a long way in reducing the model complexity and the model variance. λ is generally optimized using cross validation.

When the square of the l^2 norm is used as the regularization term, the optimization method is called l^2 regularization. At times, the l^1 norm of model parameter vectors is used as the regularization term, and the optimization method is termed l^1 regularization. l^2 regularization applied to regression problems is called ridge regression, whereas l^1 regularization applied to such regression problems is termed lasso regression.

For l^1 regularization, the preceding regression problem becomes

$$\theta^* = \underbrace{Arg\ Min}_{\theta}\ L(\theta) = \|X\theta - Y\|_2^2 + \lambda\|\theta\|_1$$

Ridge regression is mathematically more convenient since it has a closed-form solution whereas lasso regression doesn't have a closed-form solution. However, lasso regression is much more robust to outliers in comparison to ridge regression. Lasso problems give sparse solutions, so it's good for feature selection, especially when there is moderate to high correlation among the input features.

Regularization Viewed as a Constraint Optimization Problem

Instead of adding the penalty term, we can add a constraint on the magnitude of the model parameter vector to be less than or equal to some constant value. We can then have an optimization problem as follows:

$$\theta^* = argmin_\theta\ C(\theta) = \|X\theta - Y\|_2^2$$

such that $\|\theta\|_2^2 \le b$

where b is a constant.

We can convert this constrained minimization problem to an unconstrained minimization problem by creating a new Lagrangian formulation, as seen here:

$$L(\theta,\lambda) = \|X\theta - Y\|_2^2 + \lambda\left(\|\theta\|_2^2 - b\right)$$

To minimize the Lagrangian cost function per the Karush Kuhn Tucker conditions, the following are important:

- The gradient of L with respect to θ; i.e., $\nabla_\theta L(\theta,\lambda)$ should be the zero vector, which on simplification gives

  (1) $\theta = \left(X^T X + \lambda I\right)^{-1} X^T Y$

- Also at the optimal point $\lambda\left(\|\theta\|_2^2 - b\right) = 0$ and $\lambda \ge 0$

  If we consider regularization, i.e., $\lambda > 0$, then $\|\theta\|_2^2 - b = 0 \quad (2)$

As we can see from (1), θ obtained is a function of λ. λ should be adjusted such that the constraint from (2) is satisfied.

The solution $\theta = (X^T X + \lambda I)^{-1} X^T Y$ from (1) is the same as what we get from l^2 regularization. In machine-learning applications, the Lagrange multiplier is generally optimized through hyper parameter tuning or cross-validation since we have no knowledge of what a good value for b would be. When we take small values of λ the value of b increases and so does the norm of θ, whereas larger values of λ provide smaller b and hence a smaller norm for θ.

Coming back to regularization, any component in the cost function that penalizes the complexity of the model provides regularization. In tree-based models, as we increase the number of leaf nodes the complexity of the tree grows. We can add a term to the cost function that is based on the number of leaf nodes in the tree and it will provide regularization. A similar thing can be done for the depth of the tree.

Even stopping the model-training process early provides regularization. For instance, in gradient descent method the more iterations we run the more complex the model gets since with each iteration the gradient descent tries to reduce the cost-function value further. We can stop the model-learning process early based on some criteria, such as an increase in the cost function value for the test dataset in the iterative process. Whenever in the iterative process of training the training cost-function value decreases while the test cost-function value increases, it might be an indication of the onset of overfitting, and thus it makes sense to stop the iterative learning.

Whenever training data is less in comparison to the number of parameters the model must learn, there is a high chance of overfitting, because the model will learn too many rules for a small dataset and might fail to generalize well to the unseen data. If the dataset is adequate in comparison to the number of parameters, then the rules learned are over a good proportion of the population data, and hence the chances of model overfitting go down.

Summary

In this chapter, we have touched upon all the required mathematical concepts for proceeding with machine-learning and deep-learning concepts. The reader is still advised to go through proper text books pertaining to these subjects in his or her spare time for more clarity. However, this chapter is a good starting point. In the next chapter, we will start with artificial neural networks and the basics of TensorFlow.