Principles of Artificial Intelligence

Lecture Notes

Antonio Di Stasio

Supervised Learning

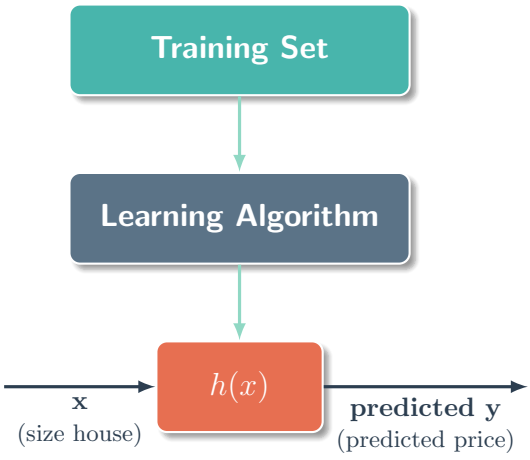
Let's begin with a straightforward example of a supervised learning problem. Imagine you have a dataset containing the living areas (in square meters) and selling prices of 100 houses in London. Your goal is to train a model that predicts a house's price based on its size.

Size (m^2)	Price (1000£)
195	400
148	330
222	369
131	232
278	540
460	760
343	399
⋮	⋮

How can we learn to predict house prices in London as a function of house size using this data?

Notations: we'll use $x^{(i)}$ to denote the "input" variables (size in this example), also called input *features*, and $y^{(i)}$ to denote the "output" or target variable that we are trying to predict (price). A pair $(x^{(i)}, y^{(i)})$ is called a *training example*, and the dataset that we'll be using to learn—a list of n training examples $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, n\}$—is called a *training set*.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function $h : X \to Y$ so that $h(x)$ is a "good" predictor for the corresponding value of y. This function h is called a *hypothesis*. Visually, the process can be represented as follows:



When the target variable we aim to predict is continuous, as in our housing example, the learning task is referred to as a **regression problem**. Conversely,

if y can take only a limited set of discrete values—such as predicting whether a dwelling is a house or an apartment based on its size—the task is known as a **classification problem**.

Linear Regression

To enhance our housing example, let's examine a more detailed dataset that also includes the number of bedrooms in each house:

Size (m^2)	# bedrooms	Price (1000£)
195	3	400
148	3	330
222	3	369
131	2	232
278	4	540
⋮	⋮	⋮

Here, the x's are two-dimensional vectors in $\mathbb{R}^2$. For instance, $x_1^{(i)}$ is the size of the i-th house in the training set, and $x_2^{(i)}$ is its number of bedrooms.

To perform supervised learning, we must decide how we're going to represent functions/hypotheses h in a computer. As an initial choice, let's say we decide to approximate y as a linear function of x:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

where the θ_i's are parameters (also called weights). To simplify our notation, we introduce the convention of letting $x_0 = 1$ so that:

$$h(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x.$$

Now, given a training set, how do we pick, or learn, the parameters θ? One reasonable method is to make $h(x)$ close to y for the training examples we have. To formalize this, we define the cost function:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2.$$

This is the familiar least-squares cost function that gives rise to the ordinary least squares regression model.

Gradient Descent

We want to choose θ to minimize $J(\theta)$. To do so, we use the **gradient descent algorithm**, which starts with some initial θ (e.g., $\theta = \overrightarrow{0}$), and repeatedly updates:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta).$$

(This update is performed simultaneously for all values of $j = 0, \ldots, d$.) Here, α is known as the **learning rate**. This algorithm naturally follows the direction of the steepest decrease in J, iteratively refining the parameters.

To implement this algorithm, we need to compute the partial derivative term on the right-hand side. Let's first derive it for the case where we have only a single training example (x, y), allowing us to omit the sum in the definition of J. We obtain:

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_\theta(x) - y)^2 \\
&= 2 \cdot \frac{1}{2} (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_\theta(x) - y) \\
&= (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^{m} \theta_i x_i - y \right) \\
&= (h_\theta(x) - y) \cdot x_j
\end{aligned}$$

For a single training example, this gives the update rule:

$$\theta_j := \theta_j - \alpha (h_\theta(x^{(i)}) - (y^{(i)})) \cdot x_j^{(i)}.$$

Now, to generalize it to the entire training set we can devise the following algorithm:

Algorithm 1 Gradient Descent

```
Initialize θ_j = 0 for all j
repeat
  for each j = 0, 1, ..., n do
    θ_j := θ_j + α Σ_{i=1}^{m} (y^(i) − h_θ(x^(i))) x_j^(i)
  end for
until convergence
```

This approach processes all examples in the training set at each iteration and is known as batch gradient descent. Although gradient descent can sometimes get trapped in local minima, the specific optimization problem in linear regression does not have this issue, as it has a single global optimum with no local optima. Consequently, gradient descent is guaranteed to converge to the global minimum, provided that the learning rate α is appropriately chosen. This is because J is a convex quadratic function.

There is an alternative to gradient descent that also works very well. Consider the following algorithm:

Algorithm 2 Stochastic Gradient Descent

```
Initialize θ_j = 0 for all j
repeat
  for each training example (x^(i), y^(i)) do
    for each j = 0, 1, ..., n do
      θ_j := θ_j + α(y^(i) − h_θ(x^(i)))x_j^(i)
    end for
  end for
until convergence
```

In this algorithm, we iterate through the training set multiple times, updating the parameters each time we encounter a training example based on the gradient of the error with respect to that single example. This method is known as **stochastic gradient descent**. Unlike gradient descent, which requires scanning the entire training set before making a single update—a computationally expensive process when m is large—stochastic gradient descent begins adjusting the parameters immediately and continues to refine them with each processed example. Often, stochastic gradient descent brings θ "close" to the optimal value much faster than batch gradient descent. However, note that it may never fully "converge" to the minimum, as the parameters θ tend to oscillate around the minimum of $J(\theta)$. Nevertheless, in practice, most of these values serve as sufficiently accurate approximations of the true minimum. Due to these advantages, particularly when dealing with large training sets, stochastic gradient descent is often preferred over gradient descent.

Logistic Regression

In this section we talk about the classification problem. This is just like the regression problem, except that the values y we now want to predict take on only a small number of discrete values. For now, we will focus on the binary classification problem in which y can take on only two values, 0 and 1. For instance, if we are trying to build a spam classifier for email, then $x^(i)$ may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. 0 is also called the negative example, and 1 the positive exaple, and they are sometimes also denoted by the symbols "-" and "+." Given $x^(i)$, the corresponding $y^(i)$ is also called the label for the training example.

Logistic regression is a statistical method used for binary classification problems, where the outcome variable can take one of two possible values, typically 0 or 1. Unlike linear regression, which predicts continuous values, logistic regression models the probability that a given input belongs to a particular class. This is done using the **logistic (sigmoid) function**, which maps real-valued inputs into a range between 0 and 1.

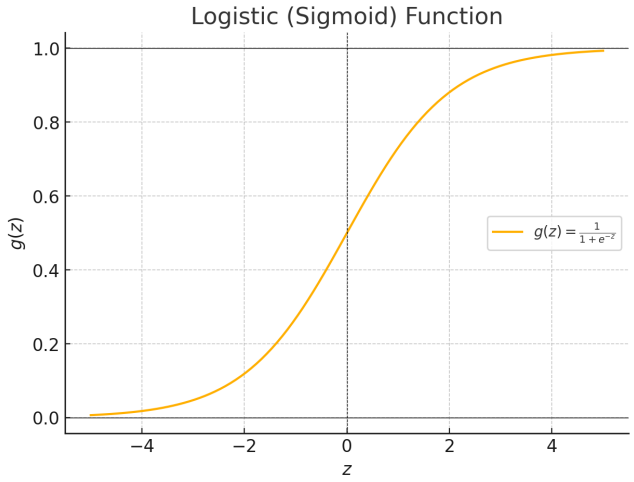
The logistic function, also known as the sigmoid function, is defined as:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

where:

$$g(z) = \frac{1}{1 + e^{-z}}$$

Here is a plot of the function $g(z)$:



The function $g(z)$ tends towards 1 as $z \to \infty$ and tends towards 0 as $z \to -\infty$. Moreover, $g(z)$, and hence $h(x)$, is always bounded between 0 and 1.

We first show an useful property of the derivative of the sigmoid function:

$$g'(z) = \frac{d}{dz}\left(\frac{1}{1 + e^{-z}}\right) \tag{1}$$

$$g'(z) = \frac{1}{(1 + e^{-z})^2} \cdot e^{-z} \tag{2}$$

$$= g'(z) = \left(\frac{1}{1 + e^{-z}}\right) \cdot \left(1 - \frac{e^{-z}}{1 + e^{-z}}\right) \tag{3}$$

$$= g(z)(1 - g(z)) \tag{4}$$

To fit θ in logistic regression, we use maximum likelihood estimation (MLE). Instead of minimizing squared errors like in linear regression, we model the probability of class labels using a logistic (sigmoid) function and determine θ by maximizing the likelihood of the observed data.

Let us assume that the conditional probabilities for our classification model are defined as follows:

$$P(y = 1|x; \theta) = h_\theta(x), \tag{5}$$

$$P(y = 0|x; \theta) = 1 - h_\theta(x). \tag{6}$$

This can be rewritten more compactly using:

$$p(y|x;\theta) = (h_\theta(x))^y(1-h_\theta(x))^{1-y}. \tag{7}$$

Assuming that the training examples are generated independently, we can write the likelihood function as:

$$L(\theta) = p(\overrightarrow{y}|X;\theta) = \prod_{i=1}^{n} p(y^{(i)}|x^{(i)};\theta), \tag{8}$$

which, by substituting our probability function, expands to:

$$L(\theta) = \prod_{i=1}^{n} (h_\theta(x^{(i)}))^{y^{(i)}}(1-h_\theta(x^{(i)}))^{1-y^{(i)}}. \tag{9}$$

To simplify optimization, we maximize the log-likelihood function:

$$\ell(\theta) = \log L(\theta) = \sum_{i=1}^{n} \left[y^{(i)} \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)})) \right]. \tag{10}$$

How do we maximize the likelihood? Similar to our derivation in the case of linear regression, we can use gradient ascent. Written in vectorial notation, our updates will therefore be given by $\theta = \theta + \alpha \frac{\partial}{\partial\theta}\ell(\theta)$. (Note the positive rather than negative sign in the update formula, since we're maximizing, rather than minimizing, a function now.) Let's start by working with just one training example (x, y), and take derivatives to derive the stochastic gradient ascent rule:

$$\frac{\partial}{\partial\theta_j}\ell(\theta) = \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)} \right) \frac{\partial}{\partial\theta_j}\theta^T x \tag{11}$$

$$= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)} \right) g(\theta^T x)(1-g(\theta^T x))\frac{\partial}{\partial\theta_j}\theta^T x \tag{12}$$

$$= \left(y(1-g(\theta^T x)) - (1-y)g(\theta^T x) \right) x_j \tag{13}$$

$$= (y - h_\theta(x))x_j \tag{14}$$

Above, we used the fact that $g'(z) = g(z)(1-g(z))$. This therefore gives us the stochastic gradient ascent rule:

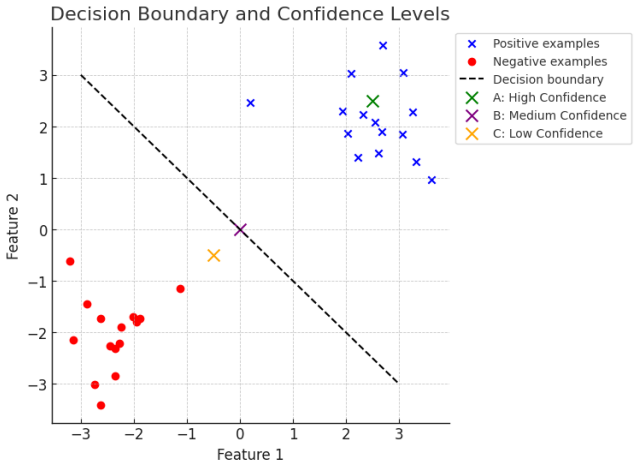
$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)} \tag{15}$$

Support Vector Machines

Support Vector Machines (SVMs) are powerful supervised learning models for classification and regression. This chapter explains their core principles, focusing on maximizing classification confidence through margin optimization.

Margin Intuition

In classification, confidence is tied to how far data points lie from the decision boundary. For a logistic regression model $h_\theta(x) = g(\theta^T x)$, predictions for $y = 1$ occur when $\theta^T x \geq 0$. Larger values of $\theta^T x$ imply higher confidence. Similarly, negative predictions ($y = 0$) gain confidence as $\theta^T x$ becomes more negative. The goal is to find parameters that ensure correct and confident predictions across all training examples.



For a geometric interpretation, we can say that points far from the decision boundary (the straight line given by the equation $\theta^T x = 0$) allow confident predictions, while those near the boundary (e.g., point C) are sensitive to small changes. Maximizing the distance (margin) between the decision boundary and the closest points improves the correctness and confidence of the predication on the training examples.

Notation

To facilitate our discussion on SVM, we first introduce a new notation for classification. We consider a linear classifier for a binary classification problem with labels y and features x. From this point forward, we denote the class labels as $y \in \{-1, 1\}$ instead of $\{0, 1\}$. Additionally, rather than parameterizing our linear classifier with the vector θ, we use the parameters w and b, and express our classifier as

$$h_{w,b}(x) = g(w^T x + b).$$

Here, we define $g(z) = 1$ if $z \geq 0$, and $g(z) = -1$ otherwise. The notation using w and b allows us to explicitly separate the intercept term b from the other parameters. Additionally, we no longer follow the previous convention of including an extra coordinate $x_0 = 1$ in the input feature vector. Consequently, the intercept term b now corresponds to the previously used θ_0, while w represents the vector $[\theta_1 \ \ldots \ \theta_d]^T$.

Functional Margin

For a training example $(x^{(i)}, y^{(i)})$, the **functional margin** is:

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b)$$

A large positive value indicates high confidence for $y^{(i)} = 1$; a large negative value indicates confidence for $y^{(i)} = -1$. Moreover, if $y^{(i)}(w^T x^{(i)} + b) > 0$, then our prediction on this example is correct. Hence, a large functional margin represents a confident and a correct prediction.

Given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, n\}$, we also define the function margin of (w, b) with respect to S as the smallest of the functional margins of the individual training examples. Denoted by $\hat{\gamma}$, this can therefore be written:

$$\hat{\gamma} = \min_{i=1,\ldots,n} \hat{\gamma}^{(i)}.$$

For a linear classifier with the choice of g as defined above (taking values in $\{-1, 1\}$), there is a specific limitation of the functional margin that makes it an unreliable measure of confidence. Given our choice of g, we observe that if we replace w with $2w$ and b with $2b$, then since

$$g(w^T x + b) = g(2w^T x + 2b),$$

this transformation does not affect $h_{w,b}(x)$ in any way. In other words, g, and consequently $h_{w,b}(x)$, depends only on the sign of $w^T x + b$ and not on its magnitude. However, scaling (w, b) to $(2w, 2b)$ also results in the functional margin being scaled by a factor of 2. Therefore, we need a more reliable measure to achieve better predictions while also accounting for the geometric distance.

Geometric Margin

The **geometric margin** normalizes the functional margin by the weight vector's magnitude $\|w\|$:

$$\gamma^{(i)} = \frac{\hat{\gamma}^{(i)}}{\|w\|} = y^{(i)} \left(\frac{w^T x^{(i)} + b}{\|w\|} \right)$$

This measure is invariant to parameter scaling, making it a reliable indicator of true model confidence.

Finally, given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, n\}$, we also define the geometric margin of (w, b) with respect to S to be the smallest of the geometric margins on the individual training examples:

$$\gamma = \min_{i=1,\ldots,n} \gamma^{(i)}.$$

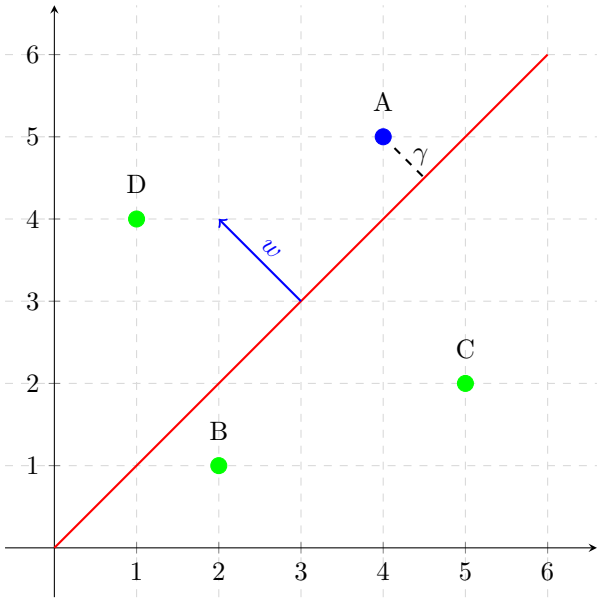


Figure 1: Geometric margin intuition. The decision boundary is shown in red, the vector w in blue (aligned with point A), and the distance γ from point A to the boundary. Additional points B, C, and D are shown in green.

Optimal Margin Classifier

To maximize the minimum geometric margin across all training examples, we solve the following optimization problem:

$$\max_{w,b,\gamma} \gamma \quad \text{s.t.} \quad \frac{y^{(i)}(w^T x^{(i)} + b)}{\|w\|} \geq \gamma \ \ \forall i$$

i.e., our goal is to maximize γ while ensuring that each training example has a functional margin of at least γ.

The problem formulation, however, is **non-convex**, making it challenging to solve directly using standard optimization techniques. A problem is non-convex when its objective function or feasible region does not form a convex set, meaning that there may exist multiple local optima rather than a single global optimum. In our case, the non-convexity arises due to the dependence on $\|w\|$,

which introduces complications in optimization. Standard convex optimization techniques, such as quadratic programming (QP), rely on convexity to ensure that any local minimum is also a global minimum. Without this property, solving the problem efficiently becomes much more difficult.

To address this issue, we reformulate the problem by normalizing the functional margin and eliminating explicit dependence on $\|w\|$:

$$\max_{\hat{\gamma},w,b} \quad \frac{\hat{\gamma}}{\|w\|}$$

$$\text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq \hat{\gamma}, \quad i = 1, \ldots, n$$

Here, we maximize $\hat{\gamma}/\|w\|$ while ensuring that all functional margins are at least $\hat{\gamma}$. Since the geometric and functional margins are related by $\gamma = \hat{\gamma}/\|w\|$, this formulation maintains our goal of maximizing the margin. However, the objective function $\frac{\hat{\gamma}}{\|w\|}$ remains non-convex, making it difficult to optimize using standard solvers.

To further simplify the problem, we leverage the fact that we can rescale w and b without affecting the classification boundary. By introducing a scaling constraint, we fix the functional margin for the training set to 1, i.e., $\hat{\gamma} = 1$. Since multiplying w and b by a constant scales the functional margin accordingly, this constraint can always be satisfied by appropriately rescaling w and b. Substituting $\hat{\gamma} = 1$ into our formulation and recognizing that maximizing $\hat{\gamma}/\|w\| = 1/\|w\|$ is equivalent to minimizing $\|w\|^2$, we arrive at the following convex optimization problem:

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

$$\text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, n$$

This reformulation results in an optimization problem with a convex quadratic objective and linear constraints, making it solvable using standard quadratic programming (QP) techniques. The solution to this problem defines the **optimal margin classifier**. Commercial QP solvers can efficiently handle such problems.

While we could stop here, we will now explore Lagrange duality, which provides an alternative formulation of our optimization problem. The dual form is particularly useful for efficiently handling high-dimensional feature spaces using kernels. Additionally, it enables the development of more efficient algorithms than general-purpose QP solvers.

Optimal Margin Classifiers: The Dual Form

Previously, we posed the following (primal) optimization problem for finding the optimal margin classifier:

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

$$\text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, n$$

In order to solve the optimization problem we need to use the method of Lagrangian multipliers α_i for constraints which transforms the primal problem into a dual form. The Lagrangian becomes:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i \left[y^{(i)}(w^T x^{(i)} + b) - 1 \right]$$

Let's find the dual form of the problem. To do so, we need to first minimize $\mathcal{L}(w, b, \alpha)$ with respect to w and b (for fixed α), which we'll do by setting the derivatives of $\mathcal{L}$ with respect to w and b to zero. We have:

Let's derive the dual form of the problem. To achieve this, we first minimize $\mathcal{L}(w, b, \alpha)$ with respect to w and b, keeping α fixed. This is done by setting the partial derivatives of $\mathcal{L}$ with respect to w and b to zero. We have that

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} = 0$$

This implies that

$$w = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)}.$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i y^{(i)} = 0.$$

If we substitute the definition of w back into the Lagrangian and simplify, we obtain:

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)} - b \sum_{i=1}^{n} \alpha_i y^{(i)}.$$

However, the last term must be zero, as shown earlier, leading to:

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)}.$$

Recall that we got to the equation above by minimizing $\mathcal{L}$ with respect to w and b. Putting this together with the constraints $\alpha_i \geq 0$ and the constraint $\sum_{i=1}^{n} \alpha_i y^{(i)} = 0$, we obtain the following dual optimization problem:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle.$$
$$\text{s.t.} \quad \alpha_i \geq 0, \quad i = 1, \ldots, n$$
$$\sum_{i=1}^{n} \alpha_i y^{(i)} = 0,$$

Kernel

The **kernel trick** is a method used in machine learning (in SVM) to implicitly map data into a high-dimensional (possibly infinite-dimensional) feature space without explicitly computing the coordinates of the data in that space.

- We often want to use a highly nonlinear feature map $\phi(\cdot)$ that transforms the original data $\mathbf{x} \in \mathbb{R}^d$ into some higher-dimensional feature space $\mathcal{F}$.

$$\phi : \mathbb{R}^d \longrightarrow \mathcal{F}.$$

- In this new space $\mathcal{F}$, linear algorithms (e.g., a linear classifier) can achieve nonlinear decision boundaries in the original input space.
- However, computing $\phi(\mathbf{x})$ *explicitly* and then taking dot products in $\mathcal{F}$ is often computationally expensive or even impossible (especially if $\mathcal{F}$ is infinite-dimensional).
- Instead, we rely on a **kernel function** $K(\mathbf{x}, \mathbf{y})$, which directly computes the dot product in the feature space:

$$K(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle.$$

- By cleverly designing $K(\mathbf{x}, \mathbf{y})$ to perform this dot product, we bypass the need to explicitly compute $\phi(\mathbf{x})$ or work in $\mathcal{F}$.

Hence, the kernel trick *implicitly* uses very high-dimensional or infinite-dimensional expansions of the data, while keeping all computations in the original space through $K(\mathbf{x}, \mathbf{y})$.

The Kernel Trick in SVM

Consider the new formulation of the optimization problem in dual form by introducing inner products. To make a prediction we need fist to learn the α_i's by solving the dual optimization problem. Then, we can make a prediction by comuting:

$$h_{w,b}(x) = g\Big(\sum_{i=1}^{n} \alpha_i \, y_i \, \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}) \rangle + b \Big).$$

Notice that all occurrences of ϕ *only* appear inside dot products of the form $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}) \rangle$. Therefore, we never need the explicit coordinates $\phi(\mathbf{x}_i)$ or $\phi(\mathbf{x})$.

Instead, we *replace* $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}) \rangle$ with $K(\mathbf{x}_i, \mathbf{x})$. Hence, the classifier becomes

$$h_{w,b}(x) \;=\; g\Big(\sum_{i=1}^{n} \alpha_i \, y_i \, K(\mathbf{x}_i, \mathbf{x}) + b \Big).$$

which is computationally feasible as long as we have a kernel function K that is simple to evaluate, *even if* $\mathcal{F}$ is an infinite-dimensional space.

Popular Kernels

Linear Kernel

$$K(\mathbf{x}, \mathbf{y}) \;=\; \mathbf{x}^\top \mathbf{y}.$$

Interpretation: This is just the standard inner product in $\mathbb{R}^d$. Using the linear kernel in an SVM, for instance, yields a *linear* decision boundary in the original input space.
When to Use: Good starting point if you suspect linear separability or if interpretability is crucial.

Polynomial Kernel

$$K(\mathbf{x}, \mathbf{y}) \;=\; (\gamma \, \mathbf{x}^\top \mathbf{y} + c)^p.$$

- **Parameters:**
  - p is the polynomial degree, controlling the *complexity* of the decision boundary. Higher p can model increasingly complex relationships.
  - γ (gamma) is a scale factor for the inner product $\mathbf{x}^\top \mathbf{y}$. This parameter controls how quickly similarity decays with distance.
  - c is a constant term added to the inner product before exponentiation. It shifts the kernel, often used to trade off higher-order vs. lower-order terms.
- **Interpretation:** Expands the feature space to include all monomials of degree up to p. For example, with $p = 2$, you include cross-terms like $x_i x_j$.
- **Typical Use Cases:** Situations where you suspect polynomial relationships but do not want to construct polynomial features explicitly.

Gaussian (RBF) Kernel

$$K(\mathbf{x}, \mathbf{y}) \;=\; \exp\!\left(-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2} \right) \;=\; \exp(-\gamma \|\mathbf{x} - \mathbf{y}\|^2),$$

where $\gamma = \frac{1}{2\sigma^2}$.

- **Parameters:**
  - σ (sigma) is the length-scale parameter.
  - $\gamma = 1/(2\sigma^2)$ is also commonly used (e.g. in scikit-learn).
- **Interpretation:** Measures similarity via a Gaussian function of the distance between points. Nearby points in input space have a large kernel value (very similar), while distant points have small kernel value.
- **Effects of Parameters:**
  - If σ (or equivalently $1/\gamma$) is too large, the kernel varies slowly, and the model may underfit (looking almost linear).
  - If σ is too small, the kernel is sharply peaked: points need to be *very close* to be considered similar, which can lead to overfitting.
- **Commonly Used:** One of the most popular kernels for SVMs and Gaussian Processes due to its flexibility and smoothness.