Machine Learning

Christian Rossi

Academic Year 2023-2024

Abstract

The course will cover several topics, starting with an introduction to basic concepts. Learning theory will be explored, including the bias-variance tradeoff, Union and Hoeffding bounds, VC dimension, worst-case online learning, and practical advice on using learning algorithms effectively.

In Supervised Learning, key areas of focus include the Supervised Learning setup, LMS, logistic regression, perceptron, the exponential family, and kernel methods such as Gaussian Processes, and Support Vector Machines. Additionally, topics like model selection, feature selection, ensemble methods, and strategies for evaluating and debugging learning algorithms will be addressed.

The course will also delve into Reinforcement Learning and control, examining Markov Decision Processes, Bellman equations, value iteration, policy iteration, Temporal Difference, SARSA, Q-Learning, value function approximation, and the Multi-Armed Bandit problem.

Contents

CHAPTER 1

Introduction

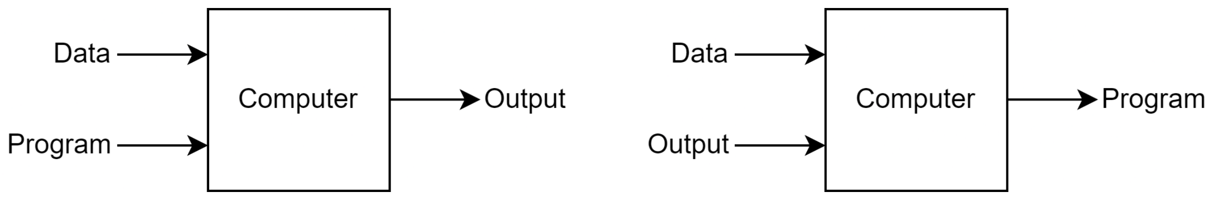
1.1 Machine Learning

Definition (*Learning*)**.** A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if it improves with experience E.

Machine Learning derives knowledge from experience and induction.

In Machine Learning, we depend on computers to make informed decisions using new, unfamiliar data. Designing a comprehensive set of meaningful rules can prove to be exceedingly difficult. Machine Learning facilitates the automatic extraction of relevant insights from historical data and effectively applies them to new datasets.

The objective is to automate the programming process for computers, acknowledging the bottleneck presented by writing software. Instead, our aim is to utilize the data itself to accomplish the required tasks.



(a) Programming (b) Machine Learning

Figure 1.1: Difference between programming and Machine Learning

Machine Learning paradigms can be categorized into three main types:

- *Supervised Learning*: involves labeled data and direct feedback, aiming to predict outcomes or future events.
- *Unsupervised Learning*: operates without labeled data or feedback, focusing on discovering hidden structures within the data.
- *Reinforcement Learning*: centers around a decision-making process, incorporating a reward system to learn sequences of actions.

1.2 Supervised Learning

Supervised Learning encompasses several distinct tasks:

- *Classification*: this involves assigning predefined categories or labels to data points based on their features. The model is trained on labeled data, learning patterns to predict the class labels of new data points.
- *Regression*: the goal here is to predict continuous numerical values based on input features, as opposed to discrete class labels in classification. The model learns a function mapping input features to output values.
- *Probability estimation*: this task predicts the likelihood of certain events or outcomes occurring, often used to gauge the confidence of model predictions.

Formally, in Supervised Learning, a model learns from data to map known inputs to known outputs. The training set is denoted as $\mathcal{D} = \{\langle x, t\rangle\}$, where $t = f(x)$, with f representing the unknown function to be determined using Supervised Learning techniques.

Various techniques can be employed for Supervised Learning, including linear models, artificial neural networks, Support Vector Machines, and decision trees.

1.3 Unsupervised Learning

Unsupervised Learning encompasses two main tasks:

- *Clustering*: in this task, the objective is to group similar data points together based on their features, without predefined labels. The goal is to uncover underlying patterns or structures within the data. Clustering algorithms segment the data into clusters or groups, where data points within the same cluster exhibit greater similarity compared to those in different clusters.
- *Dimensionality reduction*: this task involves reducing the number of input variables or features in a dataset while retaining essential information. This is often done to address the curse of dimensionality, enhance computational efficiency, and mitigate overfitting risks in models. Dimensionality reduction techniques aim to transform high-dimensional data into a lower-dimensional representation while preserving most relevant information.

Formally, in Unsupervised Learning, computers learn previously unknown patterns and efficient data representations. The training set is defined as $\mathcal{D} = \{x\}$, where the goal is to find a function f that extracts a representation or grouping of the data.

Various techniques are used for Unsupervised Learning, including k-means clustering, self-organizing maps, and principal component analysis.

1.4 Reinforcement Learning

Reinforcement Learning encompasses several key approaches:

- *Markov Decision Process*: a mathematical framework for modeling decision-making, involving states, actions, transition probabilities, and rewards. The goal is to find a policy that maximizes cumulative rewards while considering uncertainty.

- *Partially Observable Markov Decision Process*: an extension of Markov Decision Process where the current state is uncertain and must be inferred from observations. The objective remains the same, but the agent maintains a belief over possible states based on observations.
- *Stochastic games*: models for decision-making with multiple agents, where outcomes depend on actions and random factors. Players aim to optimize strategies considering other players' actions and uncertainties.

In Reinforcement Learning, the computer learns the optimal policy based on a training set $\mathcal{D}$ containing tuples $\langle x, u, x', r\rangle$, where x is the input, u is the action, x' is the resulting state after the action, and r is the reward. The policy Q^* is defined to maximize $Q^*(x, u)$ over actions u for each state x in the training set.

Various techniques such as Q-Learning, SARSA, and fitted Q-iteration are used to find this optimal policy.

CHAPTER 2

Supervised Learning

2.1 Introduction

Supervised Learning stands as the predominant and well-established learning approach. Its core objective is to enable a computer, given a training set $\mathcal{D} = \{\langle x, t\rangle\}$, to approximate a function f that maps an input x to an output t. The input variables x, often referred to as features or attributes, are paired with output variables t, also known as targets or labels. The tasks undertaken in Supervised Learning are as follows:

- *Classification*: when t is discrete.
- *Regression*: when t is continuous.
- *Probability estimation*: when$t$ represents a probability.

Supervised Learning finds application in scenarios where:

- Humans lack the capability to perform the task directly (e.g., DNA analysis).
- Humans can perform the task but lack the ability to articulate the process (e.g., medical image analysis).
- The task is subject to temporal variations (e.g., stock price prediction).
- The task demands personalization (e.g., movie recommendation).

2.1.1 Function approximation

The process of approximating a function f from a dataset $\mathcal{D}$ involves several steps:

1. *Define a loss function* $\mathcal{L}$: this function calculates the discrepancy between f and h, a chosen approximation.
2. *Select a hypothesis space* $\mathcal{H}$: this space consists of a set of candidate functions from which to choose an approximation h.
3. *Minimize* $\mathcal{L}$ *within* $\mathcal{H}$: the goal is to find an approximation h within the hypothesis space $\mathcal{H}$ that minimizes the loss function $\mathcal{L}$.

The hypothesis space $\mathcal{H}$ can be expanded to theoretically achieve a perfect approximation of the function f. However, a significant challenge arises because the loss function $\mathcal{L}$ cannot be easily determined, primarily due to the absence of the actual function f.

2.1.2 Taxonomy

The taxonomy is as follows:

- *Parametric* or *nonparametric*: parametric methods are characterized by having a fixed and finite number of parameters, while nonparametric methods have a number of parameters that depend on the training set.
- *Frequentist* or *Bayesian*: frequentist approaches utilize probabilities to model the sampling process, whereas Bayesian methods use probability to represent uncertainty about the estimate.
- *Empirical risk minimization* or *structural risk minimization*: empirical risk refers to the error over the training set, while structural risk involves balancing the training error with model complexity.

The type of Machine Learning can be:

- *Direct*: this method involves learning an approximation of f directly from the dataset $\mathcal{D}$.
- *Generative*: in this approach, the model focuses on modeling the conditional density $\Pr(t \mid x)$ and then marginalizing to find the conditional mean:
$$\mathbb{E}\left[t \mid x\right] = \int t \Pr(t \mid x)\, dt$$
- *Discriminative*: this method models the joint density $\Pr(x, t)$, infers the conditional density $\Pr(t \mid x)$, and then marginalizes to find the conditional mean:
$$\mathbb{E}\left[t \mid x\right] = \int t \Pr(t \mid x)\, dt$$

2.2 Linear regression

The goal of regression is to approximate a function $f(\mathbf{x})$ that maps input $\mathbf{x}$ to a continuous output t from a dataset $\mathcal{D}$:
$$\mathcal{D} = \{\langle \mathbf{x}, t \rangle\} \implies t = f(\mathbf{x})$$
To perform regression, we assume the existence of a function capable of performing this mapping.

In linear regression, the function $f(\cdot)$ is modeled using linear functions. This choice is motivated by several factors:

- Linear models are easily interpretable, making them suitable for explanation.
- Linear regression problems can be solved analytically, allowing for efficient computation.
- Linear functions can be extended to model nonlinear relationships.

- More sophisticated methods often build upon or incorporate elements of linear regression.

The key components of constructing a linear regression problem include:

- *Hypothesis space*: the mapping function can be defined as:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{D-1} w_j x_j = w_0 1 + \sum_{j=1}^{D-1} w_j x_j = \sum_{j=0}^{D-1} w_j x_j = \mathbf{w}^T \mathbf{x}$$

  The parameter $w_0 = -b$ is called bias parameter. In a two-dimensional space, our hypothesis space will be the set of all points in the plane (w_0, w_1). The coordinates of each point will correspond to a line in the $(\mathbf{x}, y)$ space.

- *Loss function*: we usually employ the Sum of Squared Errors:

$$\text{SSE}(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (y(x_n, \mathbf{w}) - t_n)^2 = \frac{1}{2} \sum_{n=1}^{N} (\boldsymbol{\phi}(x_n) - t_n)^2 = \text{RSS}(\mathbf{w}) = \sum_{i=1}^{N} \epsilon_i^2$$

- *Optimization*: a closed-form optimization of the RSS, known as Least Squares, begins with the matrix representation of the loss function:

$$\text{LS}(\mathbf{w}) = \frac{1}{2}\text{RSS}(\mathbf{w}) = \frac{1}{2}(\boldsymbol{\Phi}_{\mathbf{w}} - \mathbf{t})^2$$

  To find the optimal $\mathbf{w}$, we compute the first derivative of $\text{LS}(\mathbf{w})$ and set it to zero, obtaining:

$$\hat{\mathbf{w}}_{\text{LS}} = (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \mathbf{t}$$

  The inversion of the matrix can be computationally expensive, especially for large datasets, assuming the matrix is non-singular (invertible).

  To mitigate this, stochastic gradient descent can be employed. The algorithm known as Least Mean Squares (LMS) uses the following update rule:

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \alpha \left(\mathbf{w}^{(n)} \mathbf{x}^{n+1} - t^{(n+1)} \right) \mathbf{x}^{(n+1)}$$

  The same update rule can be also applied for batches of size K:

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \frac{\alpha}{K} \left(\mathbf{w}^{(n)} \mathbf{x}^{n+1} - t^{(n+1)} \right) \mathbf{x}^{(n+1)}$$

Multiple outputs If the regression problem involves multiple outputs, meaning that $\mathbf{t}$ is not a scalar, we can solve each regression problem independently. The solution for the weight vectors for all outputs can be expressed as:

$$\hat{\mathbf{W}} = \left(\boldsymbol{\Phi}^T \boldsymbol{\Phi} \right)^{-1} \boldsymbol{\Phi}^T \mathbf{T}$$

This solution can be easily decoupled for each output k:

$$\hat{\mathbf{w}}_k = \left(\boldsymbol{\Phi}^T \boldsymbol{\Phi} \right)^{-1} \boldsymbol{\Phi}^T \mathbf{t}_k$$

An advantage of this approach is that $\left(\boldsymbol{\Phi}^T \boldsymbol{\Phi} \right)^{-1}$ only needs to be computed once, regardless of the number of outputs.

2.2.1 Basis functions

While a linear combination of input variables may not always suffice to model data, we can still construct a regression model that is linear in its parameters. This can be achieved by defining a model using non-linear basis functions, expressed as:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x})$$

Here, the components of the vector $\boldsymbol{\phi}(\mathbf{x})$ are referred to as features. These features allow for a more flexible representation of the input data, enabling the model to capture non-linear relationships between the input variables and the output.

Example:
Let's reconsider a set of data regarding individuals' weight and height, along with their completion times for a one-kilometer run:

Height (cm)	**Weight (kg)**	**Completion time (s)**
180	70	180
184	80	220
174	60	170

We can model this problem using a dummy variable and introduce the Body Mass Index (BMI) as a new feature:

Dummy variable	**Height (cm)**	**Weight (kg)**	**BMI**	**Completion time (s)**
x_0	x_1	x_2	x_3	t
1	180	70	21	180
1	184	80	23	220
1	174	60	20	170

Here, the dummy variable x_0 is always initialized to one. Now, we have the option to retain or discard the weight and height variables, considering only the BMI values for analysis.

The most commonly used basis functions in regression are:

Basis function	**Formula**
Polynomial	$\phi_j(x) = x^j$
Gaussian	$\phi_j(x) = e^{-\frac{\left(x-\mu_j\right)^2}{2\sigma^2}}$
Sigmoidal	$\phi_j(x) = \dfrac{1}{1 + e^{\frac{\mu_j - x}{\sigma}}}$

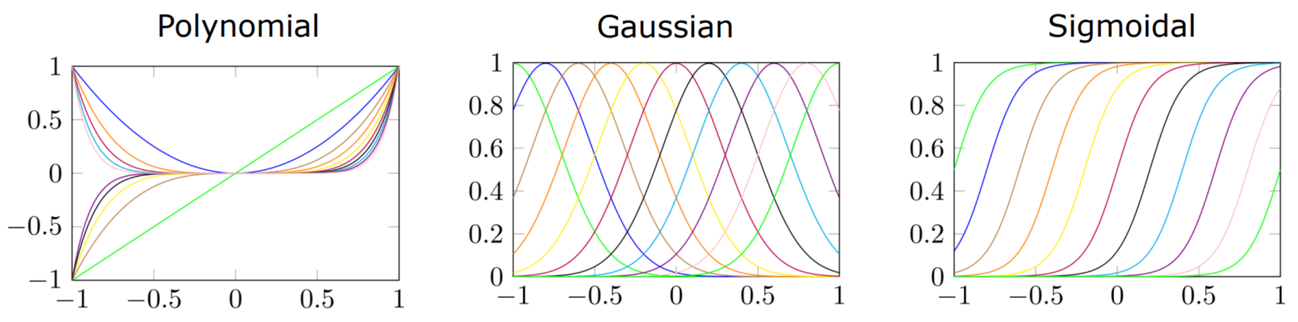


Figure 2.1: Polynomial, Gaussian, and sigmoidal basis functions

It's noteworthy that the Gaussian basis function allows for a local approximation by omitting values that are close to zero. This approach enables capturing the relationship between the input and output in a reduced input space area. As we move away from the mean, approaching zero, the values become negligible.

2.2.2 Normalization and scaling

Given a set of N samples, $\{s_1, \ldots, s_N\}$, normalization can be performed using two common methods:

- *Z-score* (normalization): scales the data based on the dataset's mean and standard deviation.. Given the mean $\bar{s}$ and the variance $S^2 = \frac{1}{N-1}\sum_{n=1}^{N}(s_n - \bar{s})^2$, the normalized value of a sample s is calculated as:

  $$s_{\text{z-score}} = \frac{s - \bar{s}}{S}$$

  This method transforms the data into a distribution with a mean of 0 and a standard deviation of 1, making it useful when working with data that needs to be compared across different scales or distributions.

- *Minmax* (feature scaling): rescales the data so that all values lie between a defined range, typically $\begin{bmatrix} 0 & 1 \end{bmatrix}$. Given the minimum value $s_{\min}$ and maximum value $s_{\max}$ in the dataset, the normalized value of a sample s is:

  $$s_{\text{Min-max}} = \frac{s - s_{\min}}{s_{\max} - s_{\min}}$$

  This method is particularly useful when the data needs to be transformed to a bounded range.

Both methods have their applications, with z-score normalization being more effective for data with outliers or differing variances, and Min-Max scaling suited for data that needs to be normalized to a specific range.

2.2.3 Regularization

A function can achieve a better approximation by increasing the degree of the polynomial used in the regression. However, increasing the polynomial degree also increases the complexity of the model parameters. To address this complexity, adjustments are needed in the loss function:

$$\mathcal{L}(\mathbf{w}) = \mathcal{L}_D(\mathbf{w}) + \lambda\mathcal{L}_W(\mathbf{w})$$

Here, $\mathcal{L}_D(\mathbf{w})$ represents the usual loss function, $\mathcal{L}_W(\mathbf{w})$ reflects model complexity (a hyper-parameter), and λ is the regularization coefficient. The model complexity loss function can be:

- *Ridge*, in which the loss function becomes:

  $$\mathcal{L}_{\text{Ridge}}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N} {\epsilon_i}^2 + \lambda\frac{1}{2}\|\mathbf{w}\|_2^2$$

  This new loss function remains quadratic with respect to $\mathbf{w}$, allowing for closed-form optimization:

  $$\hat{\mathbf{w}}_{\text{Ridge}} = \left(\lambda\mathbf{I} + \mathbf{\Phi}^T\mathbf{\Phi}\right)^{-1}\mathbf{\Phi}^T\mathbf{t}$$

  The term $\lambda\mathbf{I}$ is crucial in solving the singularity problem, as it transforms a non-singular matrix into a singular one with an appropriate choice of λ. In particular, the eigenvalues of the $\left(\lambda\mathbf{I} + \mathbf{\Phi}^T\mathbf{\Phi}\right)$ matrix must be greater or equal than λ since $\mathbf{\Phi}^T\mathbf{\Phi}$ is positive semidefinite.

- *Lasso*, in which the loss function becomes:

  $$\mathcal{L}_{\text{Lasso}}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N} \epsilon_i^2 + \lambda\frac{1}{2}\|\mathbf{w}\|_1$$

  In this case, closed-form optimization is not possible. However, Lasso typically leads to sparse regression models: when the regularization coefficient λ is large enough, some components of $\hat{\mathbf{w}}$ become equal to zero.

2.2.4 Model evaluation

The performance of the resulting model can be assessed through various metrics and statistical tests:

- *Residual Sum of Squares*: measures the discrepancy between the predicted and actual target values. A lower RSS indicates a better fit of the model to the data.

- *Mean Square Error*: average of the squared differences between the predicted values and the actual values. It is calculated as:

  $$\text{MSE}(\mathbf{w}) = \frac{\text{RSS}(\mathbf{w})}{N}$$

  where N is the number of samples. MSE penalizes larger errors more heavily due to the squaring of differences.

- *Root Mean Square Error*: square root of the MSE, giving an error metric in the same units as the target variable:

  $$\text{RMSE}(\mathbf{w}) = \sqrt{\frac{\text{RSS}(\mathbf{w})}{N}}$$

  RMSE is often easier to interpret as it provides an error measure on the same scale as the original data.

- *Coefficient of determination*: measures how well the model explains the variance in the target variable. It is calculated as:

$$R^2 = 1 - \frac{\text{RSS}(\mathbf{w})}{\text{TSS}}$$

  Here, $\text{TSS} = \sum_{n=1}^{N}(\bar{t} - t_n)^2$ is the Total Sum of Squares, and $\bar{t}$ is the mean of the target values. An R^2 close to 1 indicates a good fit, while a value near 0 suggests the model performs poorly compared to a simple mean.

- *Degrees of freedom*: represent the difference between the number of samples and the number of model parameters:

$$\text{dfe} = N - M$$

  Here, M is the number of parameters in the model.

- *Adjusted coefficient of determination*: accounts for the number of predictors in the model and adjusts for the degrees of freedom:

$$R^2_{\text{adj}} = 1 - (1 - R^2)\frac{N-1}{\text{dfe}}$$

  This metric is useful when comparing models with different numbers of predictors, as it penalizes overfitting.

Statistical tests on coeffients To determine the statistical significance of the model's parameters, hypothesis tests can be performed:

1. *Test on single coefficients*: this test examines whether each estimated weight $\hat{w}_j$ is significantly different from zero. The distribution for this test is given by:

$$t_{\text{dfe}} \sim \frac{\hat{w}_j - w_j}{\hat{\sigma}\sqrt{v_j}}$$

   Here, w_j is the true parameter, $\hat{w}_j$ is the estimated parameter, v_j is the j-th diagonal element of $(\mathbf{x}^T\mathbf{x})^{-1}$, and $\hat{\sigma}^2$ is the unbiased estimate of the variance:

$$\hat{\sigma}^2 = \frac{\text{RSS}(\hat{\mathbf{w}})}{\text{dfe}}$$

   If the test shows that the coefficient is significantly different from zero, the null hypothesis (that the coefficient is zero) is rejected.

2. *Test on overall model significance*: this test assesses the significance of the overall model by comparing it to a null model (a model with no predictors). The test uses the Fisher-Snedecor distribution:

$$F_{\text{stat}} \sim \frac{\text{dfe}}{M-1}\frac{\text{TSS} - \text{RSS}(\hat{\mathbf{w}})}{\text{RSS}(\hat{\mathbf{w}})}$$

   If F_{stat} is large, the null hypothesis (that all model coefficients are zero) is rejected, indicating that the model significantly improves prediction compared to a constant (mean) model.

2.2.5 Maximum Likelihood

We can approach regression in a probabilistic framework by defining a model that maps inputs to target values probabilistically. This allows us to express uncertainty in the predictions.

Given a regression model denoted by $y(x, \mathbf{w})$, where $\mathbf{w}$ represents the unknown parameters, we assume that the observed data $\mathcal{D}$ is generated with some inherent noise. The model provides the conditional probability of the target given the input, and we express the likelihood of the data $\mathcal{D}$ given the parameters $\mathbf{w}$ as $\Pr(\mathcal{D} \mid \mathbf{w})$.

To estimate the parameters, we seek to find the set of parameters $\mathbf{w}$ that maximizes this likelihood. This approach is known as Maximum Likelihood Estimation (ML), and the parameters are found by solving the following optimization problem:

$$\mathbf{w}_{\text{ML}} = \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathcal{D} \mid \mathbf{w})$$

Our probabilistic regression model can be written as:

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon = \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}) + \epsilon$$

Here, $y(\mathbf{x}, \mathbf{w})$ is assumed to be a linear model in terms of a set of basis functions $\mathbf{\Phi}(\mathbf{x})$, with additive noise ϵ that follows a Gaussian distribution with zero mean and variance σ^2.

Given a dataset $\mathcal{D}$ of N samples with inputs $\mathbf{x}_n$ and targets $\mathbf{t}_n$, we express the likelihood of the data $\mathcal{D}$ given the model parameters $\mathbf{w}$ as:

$$\Pr(\mathcal{D} \mid \mathbf{w}) = \Pr(\mathbf{t} \mid \mathbf{x}, \mathbf{w}, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}(t_n \mid \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_n), \sigma^2)$$

Here, $\mathcal{N}(t_n \mid \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_n), \sigma^2)$ represents the Gaussian distribution for each target, with mean $\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_n)$ and variance σ^2.

To find the Maximum Likelihood estimate $\mathbf{w}_{\text{ML}}$, we maximize the log-likelihood, which simplifies the product into a sum:

$$\mathcal{L}(\mathbf{w}) = \ln \Pr(t_n \mid \mathbf{x}_n, \mathbf{w}, \sigma^2) = -\frac{N}{2} \ln(2\pi\sigma^2) - \frac{1}{2\sigma^2} \text{RSS}(\mathbf{w})$$

Here, $RSS(\mathbf{w})$ is the Residual Sum of Squares.

The first term, $-\frac{N}{2} \ln(2\pi\sigma^2)$, is independent of $\mathbf{w}$, so we can ignore it when maximizing the log-likelihood. This leaves us with the second term, which is proportional to the residual sum of squares. Therefore, maximizing the log-likelihood is equivalent to minimizing $\text{RSS}(\mathbf{w})$.

To find $\mathbf{w}_{\text{ML}}$, we set the gradient of $\mathcal{L}(\mathbf{w})$ with respect to $\mathbf{w}$ to zero:

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = 0$$

Solving this yields the closed-form solution for the Maximum Likelihood estimate of $\mathbf{w}$

$$\mathbf{w}_{\text{ML}} = \left(\mathbf{\Phi}^T \mathbf{\Phi}\right)^{-1} \mathbf{\Phi}^T \mathbf{t}$$

This result matches the solution for the Ordinary Least Squares method, showing that the Maximum Likelihood estimate under the assumption of Gaussian noise is equivalent to minimizing the squared error. The Maximum Likelihood estimate $\mathbf{w}_{\text{ML}}$ has the smallest variance among unbiased linear estimators, according to the Gauss-Markov theorem.

2.2.6 Bayesian linear regression

Bayesian linear regression offers a probabilistic framework for modeling linear relationships by incorporating uncertainty about the model parameters, unlike traditional methods that provide only point estimates. In this approach, we treat the model parameters as random variables and update our beliefs about them as more data becomes available. The process is outlined in the following steps:

1. *Formulation of a probabilistic model*: initially, we express our prior knowledge about the model parameters probabilistically, defining a prior distribution that encapsulates assumptions about these parameters before observing any data. This prior reflects what we know or assume about the parameter values based on domain expertise or past experience.

2. *Data observation*: as we collect data, we obtain a likelihood function that measures the probability of observing the data given particular values of the model parameters.

3. *Posterior distribution calculation*: after observing the data, we use Bayes' theorem to compute the posterior distribution, which combines the prior distribution with the likelihood of the data:

$$\Pr(\text{params} \mid \text{data}) = \frac{\Pr(\text{data} \mid \text{params}) \Pr(\text{params})}{\Pr(\text{data})}$$

   The posterior distribution provides a refined belief about the model parameters after seeing the data.

4. *Prediction and decision making*: to make predictions, we use the posterior distribution by averaging over all possible parameter values weighted by their posterior probabilities. This allows for uncertainty in the predictions and enables decisions that minimize expected loss.

In Bayesian linear regression, the posterior distribution is computed by combining the prior with the likelihood of the parameters given the observed data:

$$\Pr(\mathbf{w} \mid \mathcal{D}) = \frac{\Pr(\mathcal{D} \mid \mathbf{w}) \Pr(\mathbf{w})}{\Pr(\mathcal{D})}$$

Here, $\Pr(\mathbf{w})$ is the prior distribution over the parameters $\mathbf{w}$, $\Pr(\mathcal{D} \mid \mathbf{w})$ is the likelihood of the data given the parameters, and $\Pr(\mathcal{D})$ is the marginal likelihood, ensuring normalization:

$$\Pr(\mathcal{D}) = \int \Pr(\mathcal{D} \mid \mathbf{w}) \Pr(\mathbf{w}) d\mathbf{w}$$

The mode of the posterior distribution is known as the Maximum A Posteriori (MAP) estimate, which gives the most probable parameter values given the data.

Assuming a Gaussian likelihood function allows the use of a conjugate Gaussian prior, which simplifies the Bayesian updating process. The prior is typically modeled as:

$$\Pr(\mathbf{w}) = \mathcal{N}(\mathbf{w} \mid \mathbf{w}_0, \mathbf{S}_0)$$

Here, $\mathbf{w}_0$ is the prior mean, and $\mathbf{S}_0$ is the prior covariance matrix. After observing the data, the posterior remains Gaussian:

$$\begin{cases} \Pr(\mathbf{w} \mid \mathbf{t}, \mathbf{\Phi}, \sigma^2) = \mathcal{N}(\mathbf{w} \mid \mathbf{w}_N, \mathbf{S}_N) \\ \mathbf{w}_N = \mathbf{S}_N \left(\mathbf{S}_0^{-1}\mathbf{w}_0 + \dfrac{\mathbf{\Phi}^T\mathbf{t}}{\sigma^2} \right) \\ \mathbf{S}_N^{-1} = \mathbf{S}_0^{-1} + \dfrac{\mathbf{\Phi}^T\mathbf{\Phi}}{\sigma^2} \end{cases}$$

Here, $\mathbf{w}_N$ is the posterior mean, and $\mathbf{S}_N$ is the posterior covariance matrix.

The prior mean could be:

- *Infinitely broad*: if the prior is uninformative, the covariance matrix $\mathbf{S}_0$ ends to infinity, leading to:

$$\lim_{\mathbf{S}_0 \to \infty} \mathbf{w}_N = \left(\boldsymbol{\Phi}^T \boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^T \mathbf{t} \qquad \lim_{\mathbf{S}_0 \to \infty} \mathbf{S}_N^{-1} = \frac{\boldsymbol{\Phi}^T \boldsymbol{\Phi}}{\sigma^2}$$

  This reduces the Bayesian solution to the Ordinary Least Squares (OLS) solution, and the MAP estimate becomes equivalent to the Maximum Likelihood estimate. The variance σ^2 can be estimated as:

$$\sigma^2 = \frac{1}{N-M} \sum_{n=1}^{N} \left(t_n - \hat{\mathbf{w}}^T(\phi)(\mathbf{x}_n)\right)^2$$

- *Not infinitely broad*: in cases where the prior is informative (e,g,$\mathbf{w}_0 = 0, \mathbf{S}_0 = \tau^2 \mathbf{I}$), the posterior can be expressed as:

$$\ln \Pr(\mathbf{w} \mid \mathbf{t}) = -\frac{1}{2} \sum_{i=1}^{N} \left(t_i - \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i)\right)^2 - \frac{\sigma^2}{2\tau^2} \|\mathbf{w}\|_2^2$$

  The MAP estimate coincides with the solution to Ridge regression, where the regularization parameter λ is related to the prior by $\lambda = \frac{\sigma^2}{\tau^2}$.

In Bayesian linear regression, the predictive distribution for a new data point $\mathbf{x}^*$ is given by:

$$\Pr(t \mid \mathbf{x}, \mathcal{D}) = \mathbb{E}\left[t^* \mid \mathbf{x}^*, \mathbf{w}, \mathcal{D}\right] = \int \Pr(t^* \mid \mathbf{x}^*, \mathbf{w}, \mathcal{D}) \Pr(\mathbf{w} \mid \mathcal{D})\, d\mathbf{w}$$

Under Gaussian assumptions, the predictive distribution remains Gaussian with mean and variance:

$$\mu_N(\mathbf{x}) = \boldsymbol{\phi}(\mathbf{x})^T \mathbf{W}_N \qquad \sigma_N^2(\mathbf{x}) = \sigma^2 + \boldsymbol{\phi}(\mathbf{x})^T \mathbf{S}_N \boldsymbol{\phi}(\mathbf{x})$$

As the number of data points N increases, the uncertainty in the parameters (captured by the second term) diminishes, leaving only the variance of the noise σ^2.

2.2.7 Challenges and limitations

Modeling presents challenges in ensuring our model effectively represents a wide range of plausible functions while maintaining informative priors without overly spreading out probabilities or assigning negligible values.

On the computational side, limitations arise with analytical integration, particularly in cases involving non-conjugate priors and complex models. Approaches like Gaussian approximation, Monte Carlo integration, and variational approximation become necessary for addressing these complexities and achieving accurate results.

Linear models with fixed basis functions offer several benefits:

- They permit closed-form solutions, facilitating efficient computation.
- They lend themselves to tractable Bayesian treatment, enabling principled uncertainty quantification.

- They can capture non-linear relationships by employing appropriate basis functions.

However, these models also come with several drawbacks:

- Basis functions remain static and non-adaptive to variations in the training data.
- These models are susceptible to the curse of dimensionality, particularly when dealing with high-dimensional feature spaces.

2.3 Classification

Classification involves learning an approximation of a function $f(x)$ that maps inputs x to discrete classes C_k (with $k = 1, \dots, K$) from a dataset $\mathcal{D}$:

$$\mathcal{D} = \{\langle x, C_k \rangle\} \implies C_k = f(x)$$

Various approaches to classification include:

- *Discriminant function*: modeling a parametric function that directly maps inputs to classes and learning the parameters from the data.
- *Probabilistic discriminative approach*: designing a parametric model of $\Pr(C_k \mid \mathbf{x})$ and learning the model parameters from the data.
- *Probabilistic generative approach*: modeling $\Pr(\mathbf{x} \mid C_k)$ and class priors $\Pr(C_k)$, fitting models to the data, and inferring the posterior using Bayes' rule.

In linear classification, we will use generalized linear models:

$$f(\mathbf{x}, \mathbf{w}) = f(\mathbf{x}^T\mathbf{w} + w_0)$$

Here, the function $f(\cdot)$ is not linear in $\mathbf{w}$ and partitions the input space into decision regions, with their decision boundaries. Notably, these decision boundaries are linear functions of $\mathbf{x}$ and $\mathbf{w}$, expressed as:

$$\mathbf{x}^T\mathbf{w} + w_0 = \text{constant}$$

The labels in a classification problem can be encoded in different ways, depending on the numbers of labels:

- *Two labels*: we can choose between $t \in \{0, 1\}$ and $t \in \{-1, 1\}$ depending on the specific situation. The first encoding is useful when we need to model probabilities, the second one is preferable for certain algorithms.
- *Multiple lables*: in this scenario we have K labels and the typical encoding is called 1-of-K. Here, t is a vector of length K, with a 1 in the position corresponding to the encoded class.

Two-class problem The most general formulation for a discriminant linear function in a two-class linear problem is:

$$f(\mathbf{x}, \mathbf{w}) = \begin{cases} C_1 & \text{if } \mathbf{x}^T\mathbf{w} + w_0 \geq 0 \\ C_2 & \text{otherwise} \end{cases}$$

From this formulation, we can deduce the following properties:

- The decision boundary is $y(\cdot) = \mathbf{x}^T\mathbf{w} + w_0 = 0$.
- The decision boundary is orthogonal to $\mathbf{w}$.
- The distance of the decision boundary from the origin is $\frac{w_0}{\|\mathbf{w}\|_2}$.
- The distance of the decision boundary from $\mathbf{x}$ is $\frac{y(\mathbf{x})}{\|\mathbf{w}\|_2}$.

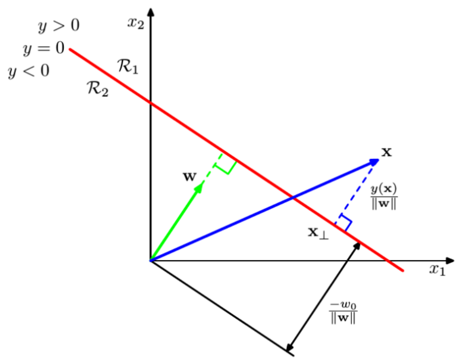


Figure 2.2: Two-class decision problem boundaries

Multiple-class problem In multiple class problems with K classes, various encoding methods can be employed:

- *One versus the rest*: this approach involves using $K - 1$ binary classifiers, where each classifier distinguishes between one class and the rest of the classes. However, this method introduces ambiguity since there may be regions mapped to multiple classes.
- *One versus one*: this method utilizes $\frac{K(K-1)}{2}$ binary classifiers, where each classifier discriminates between pairs of classes. Similar to the one versus the rest approach, this method also suffers from ambiguity.
- *Linear discriminant functions*: one solution to mitigate the ambiguity in multi-class classification is to employ K linear discriminant functions:

  $$y_k(\mathbf{x}) = \mathbf{x}^T\mathbf{w}_k + w_{k0} \qquad k = 1, \dots, K$$

  In this approach, an input vector $\mathbf{x}$ is assigned to class C_k if $y_k > y_j$ for all $j \neq k$. This method ensures that the decision boundaries are singly connected and convex.

Basis functions Up to this point, we have focused on models operating within the input space. However, we can enhance these models by incorporating a fixed set of basis functions $\boldsymbol{\phi}(\mathbf{x})$. Essentially, this involves applying a non-linear transformation to map the input space into a feature space. Consequently, decision boundaries that are linear within the feature space would correspond to nonlinear boundaries within the input space. This extension enables the application of linear classification models to problems where samples are not linearly separable.

Ordinary Least Squares Let's consider a K-class problem using a 1-of-K encoding for the target. Each class is modeled with a linear function:

$$y_k(\mathbf{x}) = \mathbf{x}^T \mathbf{w}_k + w_{k0} \qquad k = 1, \dots, K$$

In matrix notation, this can be expressed as $\mathbf{y}(\mathbf{x}) = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}}$. Given a dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{t}_i\}$ where $i = 1, \dots, N$, we can utilize the Least Squares method to determine the optimal value of $\tilde{\mathbf{w}}$, resulting in:

$$\tilde{\mathbf{w}} = \left(\tilde{\mathbf{x}}^T \tilde{\mathbf{x}}\right) \tilde{\mathbf{x}}^T \tilde{\mathbf{t}}$$

The primary challenge with employing Ordinary Least Squares in classification is that the resulting decision boundaries between regions can vary significantly based on the distribution of the data. This method may yield effective or suboptimal boundaries depending on the characteristics of the dataset.

2.3.1 Discriminant function approach

Perceptron To address the issue of poor boundaries, one approach is to utilize a model known as the Perceptron. Proposed by Rosenblatt in 1958, the Perceptron is a generalized linear model designed specifically for two-class problems. The Perceptron model is defined as:

$$f(\mathbf{x}, \mathbf{w}) = \begin{cases} +1 & \text{if } \mathbf{x}^T \mathbf{w} + w_0 \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

The Perceptron algorithm aims to determine a decision surface, also known as a separating hyperplane, by minimizing the distance of misclassified samples to the boundary. This minimization of the loss function can be achieved using stochastic gradient descent. Although simpler loss functions could theoretically be used, they are often more complex to minimize in practice.

The Perceptron loss function is expressed as:

$$\mathcal{L}_P(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^T \mathbf{x}_n t_n$$

Here, correctly classified samples do not contribute to $\mathcal{L}_P$.

Minimizing $\mathcal{L}_P$ is achieved using stochastic gradient descent:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha \mathbf{x}_n t_n$$

Since the scale of $\mathbf{w}$ does not affect the Perceptron function, the learning rate α is often set to 1. The Perceptron algorithm takes a dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{t}_i\}$ where $i = 1, \dots, N$.

Algorithm 1 Perceptron

```
1: Initialize $\mathbf{w}_0$
2: $k = 0$
3: repeat
4:     $k = k + 1$
5:     $n = k \mod N$
6:     if $\hat{t}_n \neq t_n$ then
7:         $\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{x}_n t_n$
8:     end if
9: until convergence
```

Theorem 2.3.1 (Perceptron convergence)**.** *If the training dataset is linearly separable in the feature space, then the Perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps.*

Several steps may be necessary, making it challenging to distinguish between non-separable problems and slowly converging ones. If multiple solutions exist, the one obtained by the algorithm depends on the order of the elements in the dataset.

K-Nearest Neighbors The K-Nearest Neighbors algorithm, specifically the 1-Nearest Neighbors variant, follows a discriminative approach by using the proximity of points in the feature space to make predictions for new data points. The core idea is to find the closest neighbors to predict the target label of an unseen data point.

Given a dataset $\mathcal{D} = \{(x_n, t_n)\}_{i=1}^{M}$ and a new data point $\mathbf{x}_q$, 1-NN predicts the target by finding the nearest neighbor according to the Euclidean distance:

$$i_1 \in \underset{n\in\{1,\dots,N\}}{\operatorname{argmin}} \|\mathbf{x}_q - \mathbf{x}_n\|_2$$

KNN works effectively for both regression and classification tasks, and it requires no explicit training phase; the model learns by simply querying the dataset at prediction time.

For $K > 1$, combining the targets from multiple neighbors depends on the type of task:

- *Classification*: predict the most frequent class among the K-Nearest Neighbors, with a tie-breaking rule if needed.
- *Regression*: predict the average target value among the K-Nearest Neighbors.

This approach can easily handle multiple classes without modification. In multi-class classification, the target is the mode class of the neighbors, while in multi-class regression, it is the average target value.

Note that this algorithm needs to have all the dataset stored in main memory and it is non parametric since we do not have explicit parameters to compute. Higher values for k reduces the variance.

2.3.2 Probabilistic discriminative approach

In a discriminative approach, we model the conditioned class probability directly:

$$\Pr(C_1 \mid \boldsymbol{\phi}) = \frac{1}{1 + e^{-\mathbf{w}^T\boldsymbol{\phi}}} = \sigma(\mathbf{w}^T\boldsymbol{\phi})$$

This model is commonly referred to as logistic regression (generalized linear model).

Maximum Likelihood Given a dataset $\mathcal{D} = \{\mathbf{x}_i, t_i\}$, where $i = 1, \dots, N$ and $t_i \in \{0, 1\}$, we aim to maximize the likelihood. We model the likelihood of a single sample using a Bernoulli distribution, employing the logistic regression model for conditioned class probability:

$$\Pr(t_n \mid \mathbf{x}_n, \mathbf{w}) = y_n^{t_n}(1 - y_n)^{1-t_n} \qquad y_n = \sigma(\mathbf{w}^T\boldsymbol{\phi}_n)$$

Assuming independent sampling of data in $\mathcal{D}$, we have:

$$\Pr(\mathbf{t} \mid \mathbf{x}, \mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}(1 - y_n)^{(1-t_n)} \qquad y_n = \sigma(\mathbf{w}^T\boldsymbol{\phi}_n)$$

The negative log-likelihood (also known as cross-entropy error function) serves as a convenient loss function to minimize:

$$\mathcal{L}(\mathbf{w}) = -\ln \Pr(\mathbf{t} \mid \mathbf{X}, \mathbf{w}) = -\sum_{n=1}^{N} (t_n \ln y_n + (1 - t_n) \ln(1 - y_n))$$

The derivative of the loss function yields the gradient of the loss function:

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \sum_{n=1}^{N} (y_n - t_n) \boldsymbol{\phi}_n$$

Due to the nonlinearity of the logistic regression function, a closed-form solution is not feasible. Nevertheless, the error function is convex, allowing for gradient-based optimization (online gradient descent). The convergence is asymtotically guaranteed, also in case of non linearly separable elements.

Multi class logistic regression In multi class problems, $\Pr(C_k \mid \boldsymbol{\phi})$ is modeled by applying a softmax transformation to the output of K linear functions (one for each class):

$$\Pr(C_k \mid \boldsymbol{\phi}) = \frac{e^{\mathbf{w}_k^T \boldsymbol{\phi}}}{\sum_j e^{\mathbf{w}_j^T \boldsymbol{\phi}}}$$

Similar to the two-class logistic regression and assuming 1-of-K encoding for the target, we compute the likelihood as:

$$\Pr(\mathbf{t} \mid \boldsymbol{\Phi}, \mathbf{w}_1, \ldots, \mathbf{w}_K) = \prod_{n=1}^{N} \left(\prod_{k=1}^{K} \Pr\left(C_k \mid \boldsymbol{\phi}_n\right)^{t_{nk}} \right) = \prod_{n=1}^{N} \left(\prod_{k=1}^{K} y_{nk}^{t_{nk}} \right)$$

As in the two-class problem, we minimize the cross-entropy error function:

$$L(\mathbf{w}_1, \ldots, \mathbf{w}_K) = -\ln \Pr(\mathbf{t} \mid \boldsymbol{\Phi}, \mathbf{w}_1, \ldots, \mathbf{w}_K) = -\sum_{n=1}^{N} \left(\sum_{k=1}^{K} t_{nk} \ln y_{nk} \right)$$

Then, we compute the gradient for each weight vector:

$$\frac{\partial \mathcal{L}_{\mathbf{w}_j}(\mathbf{w}_1, \ldots, \mathbf{w}_K)}{\partial \mathbf{w}} = \sum_{n=1}^{N} (y_{nj} - t_{nj}) \boldsymbol{\phi}_n$$

Replacing the logistic function with a step function in logistic regression yields the same updating rule as the Perceptron algorithm.

2.3.3 Probabilistic generative approach

The primary probabilistic generative model for classification is known as Naïve Bayes, which relies on a simplifying assumption known as the Naïve or conditional independence assumption. This assumption states that, given a class label C_k, the features x_i in the input vector $\mathbf{x}$ are conditionally independent of one another. Under this assumption, the probability of class given an input is expressed as:

$$\Pr(C_k \mid \mathbf{x}) = \Pr(C_k) \prod_{i=1}^{M} \Pr(x_j \mid C_k)$$

In Naïve Bayes classification, our goal is to predict the most likely class $y(\mathbf{x})$ for a given input $\mathbf{x}$ by maximizing the posterior probability:

$$y(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \Pr(C_k) \prod_{i=1}^{M} \Pr(x_j \mid C_k)$$

To fit a Naïve Bayes model, we typically use a logarithmic transformation of the posterior probability, resulting in a log-likelihood function that we maximize. This optimization is performed through Maximum Likelihood estimation (MLE), where both the class prior and feature likelihoods are directly estimated from the data.

It is important to note that, despite its name, Naïve Bayes is not a Bayesian method. In Bayesian analysis, priors are updated with new evidence. Here, however, we estimate priors directly from the data without subsequent updates.

Naïve Bayes, as a generative model, allows us to generate synthetic data resembling the original dataset. The process to generate a new sample is as follows:

1. Select a class $C_{\hat{k}}$ according to the estimated multinomial prior distribution with parameters $\hat{\Pr}(C_1), \ldots, \hat{\Pr}(C_K)$.
2. For each feature j, draw a sample x_j from the distribution $\mathcal{N}(\hat{\mu}_{j\hat{k}}, \hat{\sigma}^2_{j\hat{k}})$
3. Repeat this process to generate additional samples as needed.

2.3.4 Model evaluation

To assess the performance of a classifier, we can use a confusion matrix. This matrix provides a summary of the number of correctly classified and misclassified samples, offering insights into how well the model distinguishes between classes.

	Actual Class: 1	**Actual Class: 0**
Predicted Class: 1	TP	FP
Predicted Class: 0	FN	TN

Using the values from the confusion matrix, we can calculate various performance metrics:

- *Accuracy*: the fraction of total samples that are correctly classified. This is a general measure of the classifier's performance over the entire dataset.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{N}$$

- *Precision*: the fraction of samples correctly classified as positive out of all samples predicted as positive. Precision indicates how many of the predicted positive samples are actually positive.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- *Recall*: the fraction of actual positive samples that are correctly classified. Recall measures the model's ability to identify positive samples from the dataset.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- *F1 score*: the harmonic mean of precision and recall, providing a single metric that balances both. The F1 score is particularly useful when you need to find an equilibrium between precision and recall.

$$\mathrm{F1} = 2\frac{\mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}$$

Higher values for these metrics generally indicate better model performance. These measures are not symmetric; they depend on the choice of the positive class. In some applications, you may switch the positive class to obtain metrics that better reflect the classifier's predictive power in that context.

2.4 Kernel methods

Kernel methods are powerful tools for capturing nonlinear relationships in data. In nonlinear regression, the relationship between input and output variables deviates from a straightforward linear association. Similarly, in classification tasks, the class boundaries may not be linearly separable. When standard linear models fail to capture these complexities, kernel methods transform data into higher-dimensional spaces, allowing linear models to perform effectively even in nonlinear scenarios.

The transformation of data into this higher-dimensional feature space is known as feature mapping, typically represented as:

$$\Phi : x \to \phi(x)$$

However, feature mapping often encounters the curse of dimensionality, where the number of features increases exponentially with the number of input variables, making computation infeasible. Kernel methods address this by sidestepping explicit computation of the feature mapping, thus keeping the computations manageable while preserving the power of the transformation.

2.4.1 Kernel function design

The kernel function measures the similarity between two data samples and is defined as the inner product of their feature vectors:

$$k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^T \boldsymbol{\phi}(\mathbf{x}')$$

This function allows us to assess similarity without explicitly calculating the high-dimensional feature vectors. Large feature vectors can make the kernel function computationally practical, as kernel methods only require pairwise similarities, avoiding direct computation in the higher-dimensional space.

Two main strategies exist for designing kernel functions:

- *Direct construction*: designing kernel functions from scratch.

- *Composition rules*: applying a set of rules to existing kernels to construct new ones.

In both approaches, ensuring that a kernel function is valid is crucial, meaning it must correspond to a scalar product in some feature space.

Design rules According to Mercer's theorem, a valid kernel function must be continuous, symmetric, and positive semi-definite.

Theorem 2.4.1 (Mercer)**.** *Any continuous, symmetric, positive semi-definite kernel function* $k(\mathbf{x}, \mathbf{x}')$ *can be expressed as a dot product in a high-dimensional space.*

To satisfy Mercer's conditions, the Gram matrix $\mathbf{K} = \mathbf{\Phi}(\mathbf{x})^T \mathbf{\Phi}(\mathbf{x}')$, formed by evaluating the kernel on data pairs, must be positive semi-definite:

$$\mathbf{x}^T \mathbf{K} \mathbf{x}' > 0 \qquad \forall \mathbf{x}, \mathbf{x}' \neq \mathbf{0}$$

Given valid kernels $k_1(\mathbf{x}, \mathbf{x}')$ and $k_2(\mathbf{x}, \mathbf{x}')$ the following rules can be applied to design a new valid kernel:

Transformation	**Equivalence**	**Notes**
Constant	$ck_1(\mathbf{x}, \mathbf{x}')$	$c > 0$ is a constant.
Function	$f(\mathbf{x})k_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}')$	$f(\cdot)$ is any function
Polynomial	$q(k_1(\mathbf{x}, \mathbf{x}'))$	$q(\cdot)$ is a polynomial with non-negative coefficients
Exponential	$e^{k_1(\mathbf{x}, \mathbf{x}')}$	Exponential of the kernel function
Sum	$k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}')$	Addition of two kernel functions
Product	$k_1(\mathbf{x}, \mathbf{x}')k_2(\mathbf{x}, \mathbf{x}')$	Multiplication of two kernel functions
Composite	$k_3(\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}'))$	$\boldsymbol{\phi}(\mathbf{x})$ maps $\mathbf{x}$ to $\mathbb{R}^M$ k_3 is a valid kernel in $\mathbb{R}^M$
Matrix-based	$\mathbf{x}^T \mathbf{A} \mathbf{x}$	$\mathbf{A}$ is a symmetric, positive semi-definite matrix
Subspace kernels sum	$k_a(\mathbf{x}_a, \mathbf{x}'_a) + k_b(\mathbf{x}_b, \mathbf{x}'_b)$	Kernel sum over partitions of $\mathbf{x}$
Subspace kernels product	$k_a(\mathbf{x}_a, \mathbf{x}'_a)k_b(\mathbf{x}_b, \mathbf{x}'_b)$	Kernel product over partitions of $\mathbf{x}$

Kernel trick A key aspect of kernel methods is the kernel trick, where we replace terms involving $\phi(\mathbf{x})$ with equivalent expressions that rely only on $k(\mathbf{x}, \cdot)$. This allows models to compute outputs based solely on pairwise similarities between data points, making kernel methods versatile across a range of algorithms, including Ridge regression, kNN regression, the Perceptron, nonlinear Principal Component Analysis, and Support Vector Machines.

Symbolic kernels Kernel methods are not limited to real-valued vectors; they can be applied to more complex data types, such as graphs, sets, strings, and text. In such cases, the kernel function adapts to measure similarity in non-numeric data structures, enabling kernel methods to support a wide array of applications.

Generative kernels Kernel functions can also derive from probability distributions. For example, in a generative modeling context where $\Pr(\mathbf{x})$ denotes a probability distribution, a kernel function could be defined as:

$$k(\mathbf{x}, \mathbf{x}') = \Pr(\mathbf{x}) \Pr(\mathbf{x}')$$

This kernel is valid because it represents an inner product in a one-dimensional feature space where each data point $\mathbf{x}$ to $\Pr(\mathbf{x})$.

2.4.2 Kernel Ridge regression

In kernel Ridge regression, we aim to minimize a loss function that combines the Residual Sum of Squares with a regularization term to control the model complexity. The loss function, $\mathcal{L}(\mathbf{w})$, is defined as:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\mathrm{RSS} + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

Here, λ is the regularization parameter, controlling the trade-off between the fit to the data and the penalty on the magnitude of the weights $\mathbf{w}$.

To find the optimal weight vector $\mathbf{w}$, we set the gradient of $\mathcal{L}(\mathbf{w})$ with respect to $\mathbf{w}$ to zero:

$$\lambda \mathbf{w} - \mathbf{\Phi}^T \mathbf{t} + \mathbf{\Phi}^T \mathbf{\Phi} \mathbf{w} = 0$$

By rearranging and isolating $\mathbf{w}$, we can rewrite it as:

$$\mathbf{w} = \mathbf{\Phi}^T \mathbf{a}$$

Here, $\mathbf{a} = (\mathbf{K} + \lambda \mathbf{I})^{-1}\mathbf{t}$, and $\mathbf{K} = \mathbf{\Phi}\mathbf{\Phi}^T$ is the Gram matrix. The Gram matrix is an $N \times N$ matrix representing the inner products of feature vectors, indicating the pairwise similarity between training samples.

Prediction With this formulation, predictions for a new input $\mathbf{x}$ can be computed as:

$$y(\mathbf{x}) = \mathbf{k}(\mathbf{x})^T (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{t}$$

Here, $\mathbf{k}(\mathbf{x})$ is a vector with elements $k_n(\mathbf{x}) = k(\mathbf{x}_n, \mathbf{x})$ for all $\mathbf{x}_n$ in the training set $\mathcal{D}$. This formulation expresses predictions as a linear combination of the target values of the training samples, weighted by the kernel-based similarities to the new input.

Comparison In traditional Ridge regression, finding $\mathbf{w}$ requires inverting an $M \times M$ matrix $(\mathbf{\Phi}\mathbf{\Phi}^T + \lambda \mathbf{I}_M)$ which is efficient when M, the number of features, is relatively small.

In kernel Ridge regression, we invert an $N \times N$ matrix $(\mathbf{K} + \lambda \mathbf{I}_N)$, which is preferable when N the number of samples, is relatively large. This approach also avoids explicit computation of the feature map $\mathbf{\Phi}$, as the kernel function directly captures similarities between samples, resulting in a more efficient and flexible implementation.

2.4.3 Kernel k-NN regression

The k-Nearest Neighbors algorithm algorithm can be adapted for regression by predicting the output as the average of the target values of the k nearest samples in the training data. Specifically, the prediction $\hat{f}(\mathbf{x})$ for a new input $\mathbf{x}$ is given by:

$$\hat{f}(\mathbf{x}) = \frac{1}{k} \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} t_i$$

Here, $N_k(\mathbf{x})$ represents the set of the k Nearest Neighbors of $\mathbf{x}$, and t_i is the target value associated with $\mathbf{x}_i$.

While k-NN regression is simple and intuitive, it often produces noisy predictions due to the abrupt changes in neighborhood averages as the test point moves across different sample regions.

Nadaraya-Watson model The Nadaraya-Watson model, also referred to as kernel regression, smooths these abrupt changes by applying a kernel function to weight each sample based on its distance to $\mathbf{x}$. This results in a continuous, weighted average of the target values:

$$\hat{f}(\mathbf{x}) = \frac{\sum_{i=1}^{N} k(\mathbf{x}, \mathbf{x}_i) t_i}{\sum_{i=1}^{N} k(\mathbf{x}, \mathbf{x}_i)}$$

Here, $k(\mathbf{x}, \mathbf{x}_i)$ is a kernel function that assigns larger weights to samples closer to $\mathbf{x}$ and smaller weights to those further away. This approach reduces the discontinuity of predictions, offering a smoother and often more accurate regression output compared to traditional k-NN regression.

2.4.4 Gaussian Processes

A Gaussian Process defines a probability distribution over functions $y(\mathbf{x})$ such that, for any set of input points $\{\mathbf{x}_i\}_{i=1}^{N}$, the function value $\{y(\mathbf{x}_i)\}_{i=1}^{N}$ are jointly Gaussian. This property enables GPs to serve as a powerful tool for regression tasks, where the relationship between input data and output predictions is modeled probabilistically.

In particular, a GP with inputs $\mathbf{x}$ has a prior distribution over possible functions $y(\mathbf{x})$ as:

$$y(\mathbf{x}) = \mathcal{N}(\mathbf{0}, \mathbf{K})$$

Here we commonly assume a zero mean (i.e., $\boldsymbol{\mu} = \mathbf{0}$) $y(\mathbf{x})$ unless prior information suggests otherwise. The covariance matrix $\mathbf{K}$ is built from a kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$, which quantifies the similarity between pairs of input points.

Output The target variable $\mathbf{t}$ is modeled as the true process output $\mathbf{y}$, perturbed by Gaussian noise ϵ that is independent of the input:

$$\mathbf{t}_N = \mathbf{y}_N + \epsilon$$

Here, $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Thus, the conditional distribution of the observed targets given the latent function values is:

$$\Pr(\mathbf{t}_n \mid \mathbf{y}_n) = \mathcal{N}(\mathbf{t}_N \mid \mathbf{y}_N, \sigma^2 \mathbf{I}_N)$$

The prior distribution over the latent function values $\mathbf{y}_N$ is a Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{K}_n)$. Therefore, the marginal distribution of the targets $\mathbf{t}_N$ is also Gaussian:

$$\Pr(\mathbf{t}_N) = \mathcal{N}(\mathbf{t}_N \mid \mathbf{0}, \mathbf{C}_N)$$

Here, $\mathbf{C}_N = \mathbf{K}_N + \sigma^2 \mathbf{I}_N$ represents the combined effect of the kernel-based covariance and noise.

Gaussian kernel A commonly used kernel in GPs is the Gaussian (or RBF) kernel, defined as:

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{\|\mathbf{x}-\mathbf{x}'\|}{2\sigma^2}}$$

This kernel measures similarity based on Euclidean distance, where σ^2 controls the smoothness of the function.

Additionally, the Gaussian kernel can be generalized by replacing the Euclidean distance with a more general kernel-based distance measure $k(\mathbf{x}, \mathbf{x}')$, yielding:

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{k(\mathbf{x},\mathbf{x})+k(\mathbf{x}',\mathbf{x}')-2k(\mathbf{x},\mathbf{x}')}{2\sigma^2}}$$

Prediction To predict the target t_{N+1} at a new, unseen input $\mathbf{x}_{N+1}$, we use the GP framework to define the conditional predictive distribution:

$$\Pr(\mathbf{t}_{N+1}) = \mathcal{N}(\mathbf{t}_{N+1} \mid \mathbf{0}, \mathbf{C}_{N+1}) = \mathcal{N}(m(\mathbf{x}_{N+1}), \sigma^2(\mathbf{x}_{N+1}))$$

Here, $m(\mathbf{x}_{N+1}) = \mathbf{k}^T\mathbf{C}_N^{-1}\mathbf{t}$ is the predictive mean, and $\sigma^2(\mathbf{x}_{N+1}) = c - \mathbf{k}^T\mathbf{C}_N^{-1}\mathbf{k}$ is the predictive variance, providing a measure of uncertainty. The parameter $\mathbf{k}$ is the covariance vector between $\mathbf{x}_{N+1}$ and the training data points, and $c = k(\mathbf{x}_{N+1}, \mathbf{x}_{N+1})$ is the self-covariance.

Hyperparamtere estimation Although GPs are non-parametric, hyperparameters such as σ^2 (noise level) and those governing the kernel function need to be optimized or chosen. Common approaches include:

- Incorporating prior knowledge of the problem domain.
- Maximizing the log-likelihood on a validation dataset.
- Dynamically adjusting as new data becomes available.

2.4.5 Support Vector Machines

Kernel methods are a powerful tool in Machine Learning, yet they have a significant limitation: calculating the kernel function for every sample in a large training set can be computationally prohibitive. Sparse kernel methods address this by selecting a subset of the training samples, focusing only on those critical for defining decision boundaries. Two prominent examples of sparse kernel methods are Support Vector Machines (Support Vector Machines) and Relevance Vector Machines (RVMs).

Linearly separable problems To achieve optimal separation in linearly separable cases, Support Vector Machines aim to find a hyperplane that maximizes the margin. This can be formulated as:

$$\underset{\mathbf{w},b}{\operatorname{argmax}} \left\{ \min_n \left[\frac{t_n \left(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n) + b\right)}{\|\mathbf{w}\|} \right] \right\}$$

The optimization problem is often simplified by establishing a canonical hyperplane where only solutions satisfying:

$$t_n \left(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n) + b\right) = 1 \quad \forall \mathbf{x}_n \in \mathcal{S}$$

are considered. This reformulation allows us to convert the problem into a quadratic programming task:

$$\begin{aligned} \min \ & \frac{1}{2}\|\mathbf{w}\|_2^2 \\ \text{such that} \ & t_n\left(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n)+b\right) \geq 1 \quad \forall n \end{aligned}$$

Using Lagrange multipliers, we derive the dual form of this problem:

$$\begin{aligned} \max \ & \tilde{\mathcal{L}}(\boldsymbol{\alpha}) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m t_n t_m k(\mathbf{x}_n,\mathbf{x}_m) \\ \text{such that} \ & \alpha_n \geq 0 \\ & \sum_{n=1}^{N}\alpha_n t_n = 0 \quad \forall n \end{aligned}$$

The resulting classifier is represented as:

$$y(\mathbf{x}) = \sum_{n=1}^{N}\alpha_n t_n k(\mathbf{x},\mathbf{x}_n) + b$$

Only the training points with non-zero α_i values (support vectors) influence the decision boundary, and the bias term b, is computed as:

$$b = \frac{1}{|\mathcal{S}|}\sum_{\mathbf{x}_n\in\mathcal{S}}\left(t_n - \sum_{\mathbf{x}_m\in\mathcal{S}}\alpha_m t_m k(\mathbf{x}_n,\mathbf{x}_m)\right)$$

Linearly non-separable problems In real-world applications, data is often not perfectly separable. To accommodate such cases, Support Vector Machines introduce slack variables, $\xi_n \geq 0$, which allow some margin violations. This leads to the soft-margin formulation:

$$\begin{aligned} \min \ & \frac{1}{2}\|\mathbf{w}\|_2^2 + C\sum_{n=1}^{N}\xi_n \\ \text{such that} \ & t_n\left(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n)+b\right) \geq 1-\xi_n \quad \forall n \end{aligned}$$

The parameter C controls the trade-off between maximizing the margin and minimizing classification errors, with higher C values penalizing errors more heavily.

The dual form of the soft-margin problem is:

$$\begin{aligned} \max \ & \tilde{\mathcal{L}}(\boldsymbol{\alpha}) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m t_n t_m k(\mathbf{x}_n,\mathbf{x}_m) \\ \text{such that} \ & 0 \leq \alpha_n \leq C \\ & \sum_{n=1}^{N}\alpha_n t_n = 0 \quad \forall n \end{aligned}$$

In this setup, the support vectors are the samples for which $\alpha_n > 0$. If $\alpha_n < C$, then $\xi_n = 0$, indicating that the sample lies exactly on the margin boundary. Conversely, if $\alpha_n = C$, the sample may lie within the margin or be misclassified if $\xi_n > 1$.

Alternatively, this optimization problem can be reformulated using a parameter ν to control the fraction of margin violations and the number of support vectors. The equivalent formulation is:

$$\begin{aligned}
\max\ \tilde{\mathcal{L}}(\boldsymbol{\alpha}) &= -\frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m t_n t_m k(\mathbf{x}_n,\mathbf{x}_m) \\
\text{such that}\ \ 0 \leq \alpha_n &\leq \frac{1}{N} \\
\sum_{n=1}^{N}\alpha_n t_n &= 0 \\
\sum_{n=1}^{N}\alpha_n &\geq \nu \quad \forall n
\end{aligned}$$

In this case, $0 \leq \nu < 1$ is a user-defined parameter, enabling control over both the fraction of errors and the number of support vectors, ensuring that margin errors and the proportion of support vectors are limited by ν.

Training Solving the optimization problem to determine the values of α_i and b can be computationally demanding, especially for large datasets. Directly solving the Support Vector Machine optimization problem typically requires $\mathcal{O}(n^3)$ operations, where n is the number of training samples. To address this, various methods have been developed to improve efficiency:

1. *Chunking*: this approach divides the problem into smaller, more manageable chunks. Each chunk, or working set, consists of the current support vectors and a subset of samples with the highest error rates (the worst set). As iterations proceed, the size of the working set may adjust dynamically. While chunking expands the working set as necessary, it converges to the optimal solution.

2. *Osuna's methods*: a variant of chunking designed specifically for Support Vector Machine optimization, Osuna's method also employs an iterative approach but maintains a fixed working set size. Misclassified samples from the dataset replace samples in the working set during iterations, ensuring convergence while keeping the working set stable.

3. *Sequential Minimal Optimization*: reduces computational demands by iteratively optimizing only two variables at a time. This minimal working set size allows analytical solutions for each iteration, resulting in a faster convergence to the optimal solution.

For situations requiring online learning, incremental or chunking-based methods can be used to update the model continuously as new data arrives, bypassing the need for full retraining.

Multi-class While Support Vector Machines are naturally binary classifiers, they can be adapted to handle multi-class problems through the following techniques:

- *One against all*: for a k-class problem, one against all creates k binary classifiers, each distinguishing a single class from the others. During testing, the classifier with the highest margin determines the predicted class. One against all is memory-efficient but may require more training time.

- *One against one*: decomposes a k-class problem into $\frac{k(k-1)}{2}$ binary classifiers, each trained on a pair of classes. In testing, each classifier votes, and the label with the most votes is chosen. One against one offers higher performance and often yields better results.
- *Directed Acyclic Graph Support Vector Machine*: like one against one, it creates $\frac{k(k-1)}{2}$ classifiers but uses a Directed Acyclic Graph during testing, reducing the number of classifiers needed to $k-1$. This approach reduces test time while maintaining classification accuracy.

Among these methods, one against one is widely considered to perform best due to its robust pairwise decomposition, while DAG is often preferred for faster testing due to its reduced computational complexity.

CHAPTER 3

Model evaluation

3.1 Bias variance tradeoff

The bias-variance framework provides a structured approach to evaluating model performance. In this framework, we represent the data as a combination of a deterministic component and noise with zero mean and variance σ^2:

$$t = f(\mathbf{x}) + \varepsilon$$

Known process When the underlying process $f(\mathbf{x})$ generating the data is known, the correct model $y(\mathbf{x})$ for the given process can be determined using Population Risk Minimization:

$$y^*(\mathbf{x}) = \underset{y \in \mathcal{H}}{\operatorname{argmin}} \, \mathbb{E}_{t,x}[(t - y(\mathbf{x}))^2] = \int \Pr(\mathbf{x})(f(\mathbf{x}) - y(\mathbf{x}))^2 \, d\mathbf{x}$$

Unknown process When the underlying process is unknown, we use Empirical Risk Minimization:

$$\hat{y}(\mathbf{x}) = \underset{y \in \mathcal{H}}{\operatorname{argmin}} \, \frac{1}{N} \sum_{n=1}^{n} (t_n - y(x_n))^2$$

Here, the random variable $\hat{y}$ depends on the dataset.

The expected squared error can then be decomposed as follows:

$$\underbrace{\mathbb{E}_{\mathcal{D},t}\left[(t - \hat{y}(\mathbf{x}))^2\right]}_{\text{error}} = \underbrace{\sigma^2}_{\text{irreducible error}} + \underbrace{\operatorname{Var}_{\mathcal{D}}\left[\hat{y}(\mathbf{x})\right]}_{\text{variance}} + \underbrace{\mathbb{E}_{\mathcal{D}}[f(\mathbf{x}) - \hat{y}(\mathbf{x})]^2}_{\text{squared bias}}$$

In this decomposition:

- *Expected error*: averaged over the training dataset $\mathcal{D}$ and the target t.
- *Irreducible error*: unaffected by model choice or the number of samples.
- *Variance*: measures variability between models trained on different datasets, reducing as model complexity decreases or as the sample size grows. High variance leads to overfitting.

- *Squared bias*: measures the deviation between the true function and the expected learned function, depending on the hypothesis space $\mathcal{H}$. Bias generally decreases with model complexity. High bias results in underfitting.

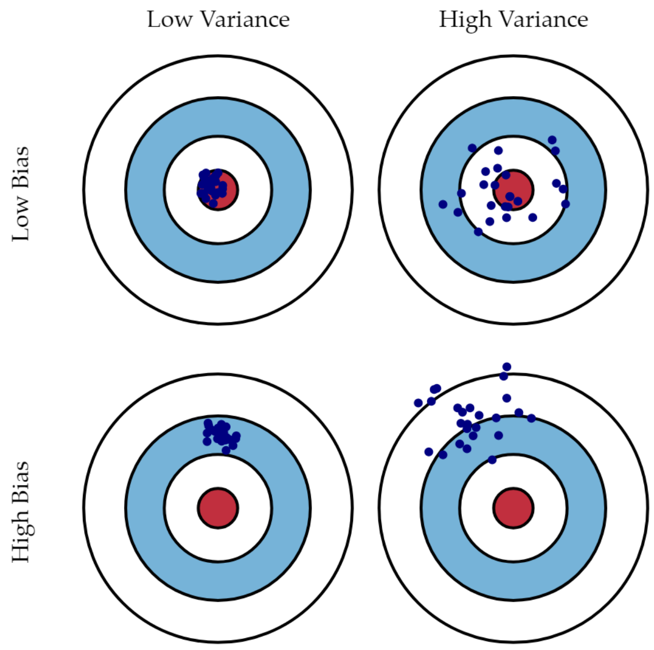


Figure 3.1: Bias-variance framework

The bias-variance decomposition shows why regularization helps reduce error on unseen data. Lasso regression tends to outperform Ridge regression when only a few features contribute to the output.

3.1.1 Training error

Given a dataset $\mathcal{D} = \{\mathbf{x}_i, t_i\}$ with $i = 1, \ldots, N$, a model is chosen based on the computed loss $\mathcal{L}$ over $\mathcal{D}$. For regression, the loss function is:

$$L_{train} = \frac{1}{N} \sum_{n=1}^{N} (t_n - y(\mathbf{x}_n))^2$$

The training error decreases as model complexity increases. However, training error does not provide an accurate estimate of the error on new data, known as the prediction error. For regression, the prediction error is represented by:

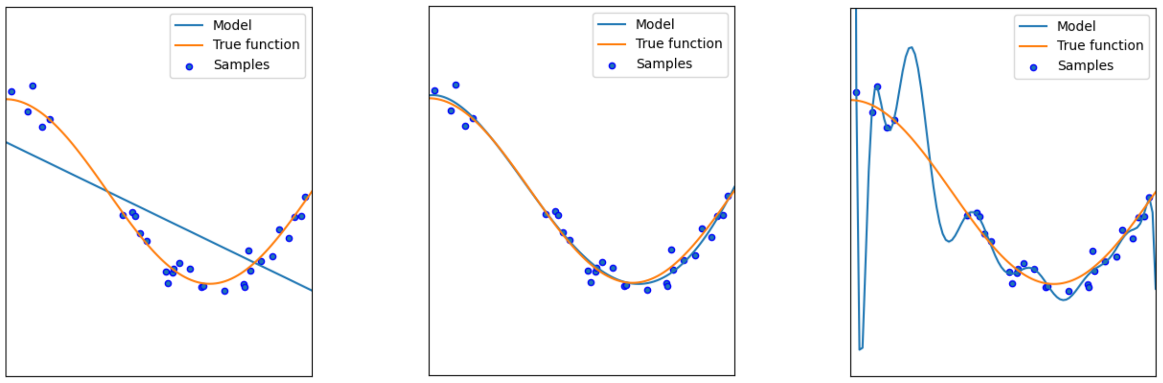
$$\mathcal{L}_{true} = \iint (t - y(\mathbf{x}))^2 \Pr(\mathbf{x}, t) \, d\mathbf{x}dt$$

Modeling the joint probability distribution $\Pr(\mathbf{x}, t)$ is often infeasible.

In practice, data is typically split into a training set and a test set. Model parameters are optimized using the training set, and prediction error is estimated using the test set. As the sample size grows, training and test errors converge. Examining train and test errors helps identify issues:

- *High bias*: when both training and test errors are high and close to each other.

- *High variance*: when training error is low, but test error gradually increases to match it.



(a) High bias (b) Balanced (c) High variance

Figure 3.2: Bias-variance balancing

When data is limited, test error may appear minimal, leading to under- or over-estimation of the prediction error. Using test error for model selection can result in overfitting to the test set. An unbiased estimate of prediction error is only achievable if the test set remains separate from training and model selection processes.

3.2 Model validation

To select the optimal model and tune hyperparameters effectively, we first divide the data into three subsets:

1. *Training set* $\mathcal{D}_{\text{train}}$: used to learn the model parameters.
2. *Validation set* $\mathcal{D}_{\text{validation}}$: used to select the best model.
3. *Test set* $\mathcal{D}_{\text{test}}$: used to evaluate the final model's performance.

A typical split is 50%-25%-25% for training, validation, and test sets, respectively. For reliable validation, the validation set must be large enough to prevent overfitting to its specific samples, which could lead to suboptimal model selection.

3.2.1 Leave-One-Out Cross Validation

In leave-one-out cross-validation, the model is trained on all samples except one $\{\mathbf{x}_i, t_i\}$, and the performance is assessed on that omitted sample. The overall prediction error estimate is the average error over all samples:

$$\mathcal{L}_{LOO} = \frac{1}{N}\sum_{i=1}^{N}\left(t_i - y_{\mathcal{D}_i}(\mathbf{x}_i)\right)^2$$

Here, $y_{\mathcal{D}_i}$ is the model trained on $\mathcal{D}$ excluding $\{\mathbf{x}_i, t_i\}$.

The $\mathcal{L}_{\text{LOO}}$ estimate is nearly unbiased, though slightly pessimistic. However, LOO-CV is computationally intensive, as it requires training N models.

3.2.2 K-Fold Cross Validation

In K-fold cross-validation, the training data $\mathcal{D}$ is split into k equally sized folds: $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k$. For each fold $\mathcal{D}_i$, the model is trained on $\mathcal{D}$ excluding $\mathcal{D}_i$, and the error is calculated on $\mathcal{D}_i$:

$$\mathcal{L}_{\mathcal{D}_i} = \frac{k}{N} \sum_{(\mathbf{x}_n, t_n) \in \mathcal{D}_i} \left(t_n - y_{\mathcal{D} \setminus \{\mathcal{D}_i\}}(\mathbf{x}_n)\right)^2$$

The prediction error is then estimated by averaging across all folds:

$$\mathcal{L}_{\text{k-fold}} = \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}_{\mathcal{D}_i}$$

The $\mathcal{L}_{\text{k-fold}}$ estimate of prediction error is slightly biased (pessimistic) but more computationally feasible than LOO-CV. Typically, k is set to 10.

3.2.3 Adjustment tecniques

Several metrics adjust the training error based on model complexity to aid in model evaluation for complex models:

Criteria	**Formula**
Mallows's C_p	$C_p = \frac{1}{N}\left(\text{RSS} + 2M\sigma^2\right)$
Akaike Information Criteria (AIC)	$\text{AIC} = -2\ln(L) + 2M$
Bayesian Information Criteria (BIC)	$\text{BIC} = -2\ln(L) + M\ln(N)$
Adjusted R^2	$A_{R^2} = 1 - \frac{\text{RSS}}{\text{TSS}} \frac{N-1}{n-m-1}$

AIC and BIC are often used when maximizing the log-likelihood. Compared to AIC, BIC imposes a stronger penalty for model complexity, favoring simpler models.

3.3 Model selection

Adding additional features increases the dimensionality of the input space exponentially. This growth not only increases computational cost but also requires more data and may introduce high variance in the model. Our objective is to select a model that minimizes prediction error by reducing variance. Achieving this requires methods that balance complexity with performance, such as:

- *Feature selection*: selecting a subset of the most relevant features to avoid unnecessary complexity.
- *Dimensionality reduction*: transforming features into a lower-dimensional space while retaining essential information.

- *Regularization*: adding penalty terms to the loss function to discourage complex models, helping to prevent overfitting.

These approaches can be combined to improve model performance, as they address complementary aspects of the model selection process.

3.3.1 Feature selection

The simplest approach to feature selection is to evaluate all possible combinations of features. However, given M features, the number of models with exactly k features to evaluate is $\binom{M}{k}$ for each subset. This exhaustive search quickly becomes computationally infeasible as M grows.

In practice, feature selection is often adapted to the type of model being used, and there are three main methods to perform feature selection: filter, embedded, and wrapper methods.

Filter methods Filter methods evaluate each feature independently, using statistical measures to assess its relevance to the target variable. The most relevant k features are then selected based on these metrics. While filter methods are computationally efficient, they may overlook interactions between features because they assess each feature individually, independent of the model.

One example of a filter method is the Pearson correlation coefficient, which measures the linear association between each feature x_j and a target y:

$$\hat{\rho}(x_j, y) = \frac{\sum_{n=1}^{N}(x_{j,n} - \bar{x}_j)(y_n - \bar{y})}{\sqrt{\sum_{n=1}^{N}(x_{j,n} - \bar{x}_j)^2}\sqrt{\sum_{n=1}^{N}(y_n - \bar{y})^2}}$$

Here, $\bar{x}_j$ and $\bar{y}$ are the mean of all the x_j and y respectively. Features with higher correlation coefficients are prioritized. Filter methods typically capture only linear relationships, though there are also extensions to detect nonlinear associations.

Embedded methods Embedded methods incorporate feature selection within the model training process itself. This approach is often used with regularized models, such as Lasso regression, which automatically drives irrelevant feature weights toward zero, effectively eliminating them. Although embedded methods are computationally efficient, they are specific to the chosen model and may not generalize to other algorithms.

Wrapper methods Wrapper methods utilize a search algorithm to find optimal feature subsets by iteratively training models on different subsets and evaluating their performance. Unlike filter methods, wrapper methods consider interactions between features. Common strategies for searching subsets include greedy algorithms such as forward selection (starting with no features and adding one at a time) and backward elimination (starting with all features and removing one at a time). Though potentially more accurate than filter methods, wrapper methods are often computationally intensive.

3.3.2 Dimensionality reduction

Dimensionality reduction seeks to reduce the number of features in the input space, but it differs from feature selection in two important ways: it utilizes all available features and projects them into a lower-dimensional space, rather than selecting a subset of the original features.

Additionally, dimensionality reduction is generally an unsupervised approach, as it does not rely on labeled data.

Several popular methods for dimensionality reduction, each with distinct strengths and use cases, include Principal Component Analysis (Principal Component Analysis), Independent Component Analysis (ICA), self-organizing maps, autoencoders, ISOMAP, and t-SNE.

Principal Component Analysis Principal Component Analysis is an unsupervised dimensionality reduction technique that performs a linear transformation on the original data to extract lower-dimensional features. The core idea of Principal Component Analysis is to find a set of orthogonal directions, or principal components, that capture the maximum variance in the data. The principal components are ranked so that the first component accounts for the highest variance, the second component for the next highest, and so on.

The steps for performing Principal Component Analysis are as follows:

1. Translate the original data $\mathbf{x}$ to $\tilde{\mathbf{x}}$ to ensure it has zero mean.
2. Compute the covariance matrix of $\tilde{\mathbf{x}}$:
$$\mathbf{C} = \tilde{\mathbf{x}}^T \tilde{\mathbf{x}}$$
The eigenvectors of $\mathbf{C}$, which correspond to the principal components of the data.
3. The eigenvectors can be computed using Singular Value Decomposition.

There are various methods for determining the number of principal components to retain:

- Retain components until the cumulative variance reaches 90%-95%. Cumulative variance is calculated as the fraction of each eigenvalue λ_i relative to the sum of all eigenvalues.
- Identify the elbow in the cumulative variance plot, where the marginal gain in explained variance begins to diminish significantly.

Principal Component Analysis is commonly used for feature extraction, data compression, and visualization in lower dimensions.

3.3.3 Regularization

Regularization is another key method for model selection, primarily aimed at reducing model complexity and preventing overfitting by penalizing larger model coefficients. Regularization techniques such as Lasso, Ridge, and Elastic Net are typically applied to linear regression models, but they can also be adapted to other types of models:

- *Lasso* (L1 regularization): adds a penalty equal to the absolute value of the coefficients. Lasso performs feature selection by forcing some coefficients to zero, effectively removing certain features from the model.
- *Ridge* (L2 regularization): adds a penalty equal to the square of the coefficients, which helps to shrink all coefficients but does not eliminate any entirely. This method is particularly useful for multicollinearity.
- *Elastic Net*: combines L1 and L2 regularization, balancing feature selection and coefficient shrinkage. Elastic Net is effective in scenarios where there are many correlated features.

Regularization not only aids in model selection by balancing model complexity and predictive accuracy but also improves generalization, particularly in high-dimensional datasets where overfitting is a concern.

3.4 Ensemble

Ensemble methods aim to reduce variance or bias (or both) by combining multiple models to improve overall predictive performance. These objectives are typically achieved through two main techniques:

- *Bagging*: reduces variance without increasing bias by training multiple models on different subsets of the data.
- *Boosting*: reduces bias by sequentially combining weak learners to create a strong model.

3.4.1 Bagging

Bagging is an ensemble technique designed to reduce model variance, making it particularly useful for high-variance, low-bias models. The steps for bagging are as follows:

1. Generate N different datasets by applying random sampling with replacement (bootstrapping) from the original training data.
2. Train a separate model (learner) on each of these datasets in parallel.

For prediction, each model is applied to a new sample, and the outputs are combined:

- In classification, a majority vote across the models is used.
- In regression, predictions are averaged.

Bagging reduces variance by averaging out the noise associated with individual models. It is particularly effective for unstable learners (models highly sensitive to small changes in the data). For such learners, bagging can improve stability and performance. However, bagging is less effective for robust learners with inherently low variance, like linear models.

	Bagging
Primary Goal	Reduces variance
Works Best With	Unstable learners (sensitive to data changes)
Noise Handling	Can be applied with noisy data
Effectiveness	Usually helps, but the difference might be small
Execution	Parallel

3.4.2 Boosting

Boosting is an iterative ensemble technique that combines weak learners sequentially to build a strong model, focusing on reducing bias. The steps in boosting are as follows:

1. Assign equal weights to all samples in the training set initially.
2. Train a weak learner (often a simple model with high bias) on the weighted dataset.
3. Calculate the error of this model on the training set.
4. Increase the weights of the misclassified samples so that the next model focuses more on these difficult cases.

5. Repeat from step 2 until a stopping criterion is met (e.g., a maximum number of learners or a desired level of accuracy).

Once all learners are trained, they are combined by weighting each model's predictions according to its accuracy. For new samples, the ensemble prediction is a weighted sum (or weighted vote) of the predictions from all the weak learners, with more accurate learners contributing more heavily.

	Boosting
Primary Goal	Reduces bias (generally without overfitting)
Works Best With	Stable learners (less sensitive to data changes)
Noise Handling	Might have problems with noisy data
Effectiveness	Not always helpful, but can make a significant difference
Execution	Sequential

3.5 Computational learning theory

We start by considering an input space $\mathcal{X}$ with M-dimensional features, with an output space denoted as $\mathcal{Y}$. We also have a joint probability $\Pr(\mathbf{x}, t)$, a loss function $\mathcal{L}$, and the hypothesis space $\mathcal{H} \subset \{h : \mathcal{X} \times \mathcal{Y}\}$

Let's suppose L has identified a hypothesis h^* that makes no error on the training data. We need to find how many training samples from $\mathcal{X}$ are required to ensure that L has learned a true concept.

Theorem 3.5.1 (No free lunch)**.** *Let $acc_G(L)$ represent the accuracy of learner L on samples not included in the training set. Let $\mathcal{F}$ be the collection of all potential concepts where $y = f(\mathbf{x})$. For any binary classifier L and any possible training set:*

$$\frac{1}{|\mathcal{F}|} \sum_{\mathcal{F}} acc_G(L) = \frac{1}{2}$$

This means that on avereage every binary classification will behave like a random guess.

This means that there is no model in Machine Learning that is superior with respect to all other models. This means also that in Machine Learning we always operate under some assumptions (such as the assumptions that at least a good approximation of the searched function is in the hypothesis space).

Corollary 3.5.1.1. *For any two learners, L_1 and L_2, if exists $f(\cdot)$ where $acc_G(L_1) > acc_G(L_2)$ then exists $f'(\cdot)$ where $acc_G(L_2) > acc_G(L_1)$.*

3.5.1 Ideal learning

Let's assume that the hypothesis space $\mathcal{H}$ contains the real function, so the learner L can obtain a null training error.

Let be $\mathcal{D}$ be the training data drawn from a stationary distribution and labeled (without noise) according to a concept we intend to learn. A binary classifier L produces a hypothesis $h \in H$ such that:

$$h^* = \operatorname*{argmin}_{h \in H} \text{error}_{\text{train}}(h)$$

We determine the training error of a hypothesis as the probability of misclassifying a sample:

$$\text{error}_{\mathcal{D}}(h) = \Pr_{x \in \mathcal{D}} [h(x) \neq c(x)]$$

However, our interest lies in the true error (probability of making a mistake on a sample) of:

$$\text{error}_{\text{true}}(h) = \Pr_{x \sim P(X)} [h(x) \neq c(x)]$$

We say that $\mathcal{H}$ overfits the training data if $\text{error}_{\text{true}} > \text{error}_{\mathcal{D}}$, but we cannot accurately bound $\text{error}_{\text{true}}$ given $\text{error}_{\mathcal{D}}$ because the training data are not independent of $\mathcal{H}$ Therefore, we require a stricter bounding of the error under additional assumptions.

Definition (*Consistent hypothesis*)**.** A hypothesis $\mathcal{H}$ is deemed consistent with a training dataset $\mathcal{D}$ of the concept c if and only if $h(x) = c(x)$ for each training sample in $\mathcal{D}$:

$$\text{consistent}(h, \mathcal{D}) \overset{\text{def}}{=} h(x) = c(x) \qquad \forall \langle x, c(x) \rangle \in \mathcal{D}$$

In other words, is an hypothesis with null training error.

Definition (*Version space*)**.** The version space, $\text{VS}_{\mathcal{H},\mathcal{D}}$, with respect to the hypothesis space $\mathcal{H}$ and the labeled dataset $\mathcal{D}$, is the subset of hypothesis in $\mathcal{H}$ consistent with $\mathcal{D}$:

$$\text{VS}_{\mathcal{H},\mathcal{D}} \overset{\text{def}}{=} \{h \in \mathcal{H} \mid \text{consistent}(h, \mathcal{D})\}$$

Thus, the version space consists of all the hypothesis spaces in which we have a null training error.

From now on, we consider only consistent learners, which always output a consistent hypothesis. This means that each time we train a Machine Learning algorithm, it will always find the function with zero training error when we have a consistent hypothesis.

If we aim to bound the $\text{error}_{\text{true}}$ of a consistent learner, we need to find a bound for all the hypothesis in $\text{VS}_{\mathcal{H},\mathcal{D}}$.

Theorem 3.5.2. *If the hypothesis space $\mathcal{H}$ is finite and $\mathcal{D}$ is a sequence of $N \geq 1$ independent random examples of some target concept c, then for any $0 \leq \varepsilon \leq 1$, the probability that $VS_{\mathcal{H},\mathcal{D}}$ contains a hypothesis error greater than ε is less than $|\mathcal{H}| e^{\varepsilon N}$:*

$$\Pr(\exists h \in \mathcal{H} \mid \mathit{error}_{\mathcal{D}}(h) = 0 \land \mathit{error}_{\mathit{true}}(h) \geq \varepsilon) \leq |\mathcal{H}| e^{\varepsilon N}$$

Practical application Let's denote δ as the probability of having $\text{error}_{\text{true}} > \varepsilon$ for a consistent hypothesis:

$$|\mathcal{H}| e^{-\varepsilon N} \leq \delta$$

By using the logaritms, we can bound both N and ε.

Definition (*Probably Approximately Correct learnable concept*)**.** A concept C is PAC-learnable by L using $\mathcal{H}$ if:

$$\forall c \in C, \text{distributions } \Pr(\mathcal{X}), 0 < \varepsilon < \frac{1}{2}, 0 < \delta < \frac{1}{2}$$

The learner L will with a probability at least $1 - \delta$ output a hypothesis $h \in \mathcal{H}$ such that:

$$\text{error}_{\text{true}}(h) \leq \varepsilon$$

In time that is polynomial in $\frac{1}{\varepsilon}, \frac{1}{\delta}, M,$ and $\text{size}(c)$.

A sufficient condition to prove PAC learnability is proving that a learner L requires only a polynomial number of training examples, and processing per example is polynomial.

3.5.2 Agnostic learning

Up to this point, we've operated under the assumption that the version space $\mathrm{VS}_{\mathcal{H},\mathcal{D}}$ is not empty, and that the learner L will consistently output a hypothesis $\mathcal{H}$ such that the error on the dataset is zero. However, in a more general scenario, an agnostic learner might output a hypothesis $\mathcal{H}$ with $\mathrm{error}_{\mathcal{D}}(h) > 0$. This means, that it will output a function that is similar to the real one, but not exactly the same.

Theorem 3.5.3. *If the hypothesis space $\mathcal{H}$ is finite and $\mathcal{D}$ is a sequence of $N \geq 1$ independent and identically distributed random variables examples of some target concept c, then for any $0 \leq \varepsilon \leq 1$, and for any learned hypothesis $\mathcal{H}$, the probability that $error_{true}(h) - error_{\mathcal{D}}(h) > \varepsilon$ is less than $|\mathcal{H}|\, e^{-2N\varepsilon^2}$:*

$$\Pr(\exists h \in \mathcal{H} \mid error_{true}(k) > error_{\mathcal{D}}(h) + \varepsilon) \leq |\mathcal{H}|\, e^{-2N\varepsilon^2}$$

Similar to previous derivations, we can establish a bound on the sample complexity:

$$N \geq \frac{1}{2\varepsilon^2}\left(\ln|\mathcal{H}| + \ln\left(\frac{1}{\delta}\right)\right)$$

Furthermore, we can also constrain the true error of the hypothesis as follows:

$$\mathrm{error}_{\mathrm{true}}(h) \leq \mathrm{error}_{\mathcal{D}}(h) + \sqrt{\frac{\ln|\mathcal{H}| + \ln\frac{1}{\delta}}{2N}}$$

Here, $\mathrm{error}_{\mathcal{D}}(h)$ describe the bias, while the other term describes the variance.

3.5.3 Vapnik-Chervonenkis dimension

The VC dimension represents the size of the subset of X for which $|\mathcal{H}|$ can ensure a zero training error, regardless of the target function.

Definition (*Dichotomy*)**.** A dichotomy of a set S of instances is defined as a partition of S into two disjoint subsets.

Definition (*Shattered*)**.** A set of instances S is said to be shattered by hypothesis space $\mathcal{H}$ if and only if for every dichotomy of S, there exists some hypothesis in $\mathcal{H}$ consistent with this dichotomy.

Definition (*VC dimension*)**.** The Vapnik-Chervonenkis dimension, $\mathrm{VC}(\mathcal{H})$, of hypothesis space $\mathcal{H}$ over instance space X, is the largest finite subset of X shattered by $\mathcal{H}$.

If an arbitrarily large set of $\mathcal{X}$ can be shattered by $\mathcal{H}$, then $\mathrm{VC}(\mathcal{H}) = \infty$.

If $|\mathcal{H}| < \infty$, then $\mathrm{VC}(\mathcal{H}) \leq \log_2(|\mathcal{H}|)$. When $\mathrm{VC}(\mathcal{H}) = d$, it implies that there are at least 2^d hypothesis in $\mathcal{H}$ to label d instances. Consequently, $|\mathcal{H}| \geq 2^d$. With a probability of at least $(1-\delta)$, every $h \in \mathcal{H}$ satisfies the following inequality:

$$\mathrm{error}_{\mathrm{true}}(h) \leq \mathrm{error}_{\mathcal{D}}(h) + \sqrt{\frac{\mathrm{VC}(\mathcal{H})\left(\ln\frac{2N}{\mathrm{VC}(\mathcal{H})} + 1\right) + \ln\frac{4}{\delta}}{N}}$$

CHAPTER 4

Reinforcement Learning

4.1 Introduction

Sequential decision-making is a field that examines how agents make a series of interconnected decisions to achieve specific goals. In such scenarios, the optimal choice of actions often depends on the context, and the correct path forward is rarely obvious. Decisions are complicated further by their long-term consequences: actions that may appear suboptimal in the short term can be strategically vital for accomplishing broader objectives.

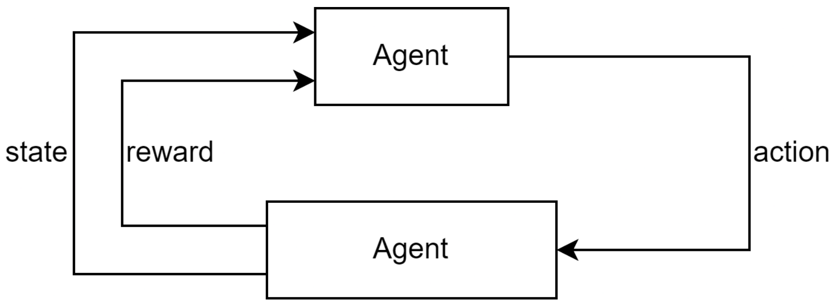


Figure 4.1: Agent-environment interface

This interaction unfolds at discrete time steps $t = 0, 1, 2, \ldots$, and follows a structured cycle:

1. *State observation and action selection*: the agent observes the current state $S_t \in \mathcal{S}$ and selects an action $A_t \in \mathcal{A}(S_t)$ based on that state.

2. *Feedback and transition*: the environment responds by providing a reward $R_{t+1} \in \mathcal{R}$ and transitioning to a new state $S_{t+1} \in \mathcal{S}$ as a result of the agent's action.

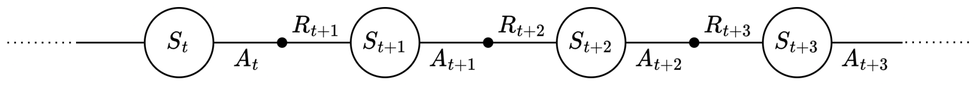


Figure 4.2: Agent-environment interaction

4.2 Markov Decision Process

To model the dynamics of a process and the ability to choose among different actions in each situation, we focus on two main problems:

1. *Prediction*: given a specific behavior (policy) for each situation, estimate the expected long-term reward starting from a particular state.
2. *Control*: learn the optimal behavior to follow in order to maximize the expected long-term reward provided by the underlying process.

A Markov Decision Process is formally defined as $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \mu, \gamma)$, where:

- *States* ($\mathcal{S}$): these represent the possible configurations the system can be in.
- *Actions* ($\mathcal{A}$): these are the choices or decisions available to the agent at each state.
- *Transition model* (P): this defines the probability distribution over the next states given the current state and action.
- *Reward function* (R): the reward function assigns a numerical value to each state-action pair representing the immediate benefit of performing action a in state s.
- *Initial distribution* (μ): this describes the probability distribution over the initial states, $\mu(s)$, indicating where the process begins.
- *Discount factor* (γ): a scalar in the range $(0, 1]$ that determines the importance of future rewards. A lower γ emphasizes immediate rewards, while a higher γ values long-term rewards more heavily.

Markov Decision Processes are grounded in the Markov property.

Property 4.2.1. *The future state s' and reward r depend solely on the current state s and action a.*

This assumption is not restrictive, as it can be interpreted as a property of the state itself.

When the Markov property holds and both the state and action spaces are finite, the process is referred to as a finite Markov Decision Process. To formally define a finite Markov Decision Process, we specify the sets of states and actions, along with the one-step dynamics.

4.2.1 Rewards

In an Markov Decision Process, an agent must prioritize long-term rewards over immediate gains to make effective decisions. The agent's goal is to maximize the cumulative future rewards, often referred to as the return:

$$G_t = f(R_{t+1} + R_{t+2} + R_{t+3} + \dots)$$

The key to success lies in maximizing the expected return, which can take various forms such as the total reward, discounted reward, or average reward.

Based on the type of task we may have a different reward function:

- *Episodic tasks*: the interaction between the agent and the environment is divided into distinct episodes. Each episode has a clear starting and ending point. For such tasks, the expected total reward is given by:

$$\mathbb{E}\left[G_t\right] = \mathbb{E}\left[\sum_{k=0}^{T} R_{t+k+1}\right]$$

- *Continuing tasks*: the agent interacts with the environment indefinitely, without a terminal state. Here, the total reward involves an infinite sequence, which may diverge. To address this, future rewards are discounted by a factor γ, where $0 < \gamma < 1$:

$$\mathbb{E}\left[G_t\right] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right]$$

The objective in Reinforcement Learning is to define the desired outcome rather than prescribing specific actions. This principle aligns with the reward hypothesis, which states that all goals can be expressed as the maximization of the expected cumulative reward.

4.2.2 Policy

An agent's behavior in a Markov Decision Process is governed by its policy, which maps states to actions. Mathematically, a policy is defined as:

$$\pi : \mathcal{S} \to \Delta(\mathcal{A})$$

The policy determines the probabilities of selecting actions in each state.

For a given policy $\pi(a \mid s)$, the following terms are derived:

$$P_\pi(s' \mid s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \Pr(s' \mid s, a) \qquad R_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) R(s, a)$$

These equations describe the transition probabilities and expected rewards under policy π. A policy fully defines the agent's decision-making process and can vary based on how actions are selected.

The policy can be:

- *Deterministic*: maps each state s to a specific action a:

$$\pi(s) = a$$

  In this case, the policy is a simple, unambiguous function that specifies exactly one action per state. Deterministic policies are often represented as lookup tables where each state corresponds to a unique action.

- *Stochastic*: assigns probabilities to actions in a given state:

$$\pi(a \mid s)$$

  Here, $\pi(a \mid s)$ represents the probability of taking action a in state s, with the sum of probabilities over all actions equal to 1. Stochastic policies are more flexible than deterministic ones, as they allow for random exploration. A deterministic policy can be seen as a special case of a stochastic policy where one action has probability 1 and all others have probability 0.

- *Markovian*: depends solely on the current state s, adhering to the Markov property. In this framework, the action selection process is memoryless, relying only on the present state rather than the history of previous states or actions. In contrast, a non-Markovian policy incorporates past interactions (or a summary of them) into decision-making, introducing dependencies beyond the current state.

By modifying the policy, different strategies can be explored, such as:

- *Myopic policies*: focus on immediate rewards, often prioritizing short-term gains.
- *Far-sighted policies*: consider long-term rewards, optimizing for cumulative future gains.

4.2.3 Reward functions

The state-value function $V_\pi(s)$ represents the expected cumulative reward starting from state s and following policy π:

$$V_\pi(s) = \mathbb{E}\left[G_t \mid S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]$$

The action-value function $Q_\pi(s, a)$ represents the expected cumulative reward starting from state s, taking action a, and then following policy π:

$$Q_\pi(s, a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]$$

This function evaluates the quality of taking a specific action in a given state under a particular policy.

Bellman expectation equation The state-value function can be expressed recursively using the Bellman expectation equation:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s)\left[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s' \mid s, a) V_\pi(s')\right]$$

In matrix form, this is written as:

$$V_\pi = R_\pi + \gamma P_\pi V_\pi$$

The solution of this equation can be found in two ways:

1. *Closed form solution*: if P_π is a finite stochastic matrix, the state-value function can be solved directly:
$$V_\pi = (I - \gamma P_\pi)^{-1} R_\pi$$
Since P_π is stochastic, the eigenvalues of $(I - \gamma P_\pi)$ lie in $[1 - \gamma, 1]$, ensuring the matrix is invertible for $0 \leq \gamma < 1$.

2. *Recursive solution*: for large state spaces where direct inversion is impractical, an iterative approach is used:
$$V_\pi = R_\pi + \gamma P_\pi V_\pi$$

Similarly, the action-value function satisfies a recursive Bellman equation:

$$Q_\pi(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s' \mid s, a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s') Q_\pi(s', a')$$

4.2.4 Policy optimality

The relationship between two policies, π and π', is defined as $\pi \geq \pi'$ if and only if the state-value function under π is greater than or equal to that under π' for all states $s \in \mathcal{S}$:

$$\pi \geq \pi' \iff V_\pi(s) \geq V_{\pi'}(s)$$

In any Markov Decision Process, there exists at least one optimal policy π^*, which is as good as or better than any other policy. This means that for every state, following π^* yields the highest possible cumulative reward. The existence of such a policy is guaranteed because, for each state, we can identify the best action to take.

However, determining the optimal policy through brute force is computationally infeasible. The number of deterministic policies grows exponentially with the size of the state and action spaces: $\mathcal{O}\left(|\mathcal{A}|^{|\mathcal{S}|}\right)$.

To overcome this challenge, we use value functions to represent the quality of states and actions. The optimal state-value function $V^*(s)$ and the optimal action-value function $Q^*(s,a)$ are defined as:

$$V^*(s) = \max_\pi V_\pi(s) \qquad Q^*(s,a) = \max_\pi Q_\pi(s,a)$$

These functions represent the best possible return achievable from a state or state-action pair under any policy.

Functions optimality The optimal value functions satisfy recursive relationships known as the Bellman optimality equations:

$$V^*(s) = \max_a \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s' \mid s,a) V^*(s') \right\}$$

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s' \mid s,a) \max_{a'} Q^*(s',a')$$

Policy optimality Given the optimal value functions $V^*(s)$ or $Q^*(s,a)$, the optimal policy π^* can be derived as:

$$\pi^*(s) = \operatorname*{argmax}_a \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s' \mid s,a) V^*(s') \right\}$$

While these formulations provide a direct way to compute the optimal policy, solving for $V^*(s)$ or $Q^*(s,a)$ can be computationally expensive in large state and action spaces. This is due to the need to evaluate all states, actions, and transitions, which may not be feasible in practice.

4.3 Dynamic Programming

Dynamic Programming provides a systematic and efficient way to address this challenge by breaking down the complex problem into smaller, more manageable sub-problems. It leverages the recursive nature of the Bellman equations to iteratively compute solutions. Dynamic Programming methods rely on two core strategies to solve Markov Decision Processes: policy iteration and value iteration.

4.3.1 Policy iteration

Policy iteration is a method in Dynamic Programming that combines two alternating steps (policy evaluation and policy improvement) to iteratively refine a policy until it converges to the optimal policy, π^*.

Policy evaluation The goal of policy evaluation is to compute the value function $V_\pi(s)$ for a given policy π. This is done by solving the Bellman expectation equation with an iterative approach:

$$V_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s' \mid s,a) V_k(s') \right]$$

We start with an arbitrary value function $V_0(s)$, and at each iteration k, we update $V_k(s)$ for all $s \in \mathcal{S}$.

This process converges to $V_\pi(s)$ as $k \to \infty$. The stopping criterion is typically defined by a small threshold $\theta > 0$, which ensures that updates become negligibly small.

Policy improvement Once $V_\pi(s)$ is computed, we improve the policy by acting greedily with respect to the current value function:

$$\pi'(s) = \operatorname*{argmax}_a \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s' \mid s,a) V_\pi(s') \right\} = \operatorname*{argmax}_a Q_\pi(s,a)$$

This step determines the best action a for each state s, based on the value function $V_\pi(s)$. Two possible outcomes arise:

- $\pi' = \pi$: the policy is already optimal (π^*).
- $\pi' \neq \pi$: the new policy π' is strictly better than π.

The improvement step is guaranteed to yield a policy that is at least as good as the previous one, based on the policy improvement theorem.

Theorem 4.3.1. *For any pair deterministic policies π' and π such that:*

$$Q_\pi(s, \pi'(s)) \geq Q_\pi(s, \pi(s)) \qquad \forall s \in \mathcal{S}$$

We have that π' is better or as good as π:

$$\pi' \geq \pi$$

4.3.1.1 Policy iteration algorithm

The iterative process alternates between policy evaluation and policy improvement until the policy stabilizes:

1. *Initialization*: start with an arbitrary policy $\pi(s)$ and initialize $V(s)$.
2. *Policy evaluation*: iteratively compute $V_\pi(s)$ until convergence ($\Delta < \theta$).
3. *Policy improvement*: update $\pi(s)$ by acting greedily with respect to $V_\pi(s)$.

4. *Repeat steps 2 and 3* until the policy stabilizes.

Algorithm 2 Policy iteration

```
1: $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$    ▷ Initialization
2: repeat
3:     repeat    ▷ Policy evaluation
4:         $\Delta = 0$
5:         for each $s \in \mathcal{S}$ do
6:             $v = V(s)$
7:             $V(s) = \sum_a \pi(a \mid s) \sum_{s',r} \Pr(s', r \mid s, a) \left[r + \gamma V(s')\right]$
8:             $\Delta = \max\left(\Delta, |v - V(s)|\right)$
9:         end for
10:    until $\Delta < \theta$
11:    policy-stable = true    ▷ Policy improvement
12:    for each $s \in \mathcal{S}$ do
13:        old-action = $\pi(s)$
14:        $\pi(s) = \text{argmax}_a \sum_{s',r} \Pr(s', r \mid s, a)[r + \gamma V(s')]$
15:        if old-action $\neq \pi(s)$ then
16:            policy-stable = false
17:        end if
18:    end for
19: until policy-stable = true
20: return $V \approx v^*$ and $\pi \approx \pi^*$
```

The policy iteration algorithm is guaranteed to converge to the optimal policy π^* and the corresponding value function $V^*(s)$ in a finite number of steps. The improvement step ensures that the policy becomes progressively better, and the evaluation step refines the value function accordingly.

4.3.2 Value iteration

Value iteration is an efficient algorithm that combines elements of both policy evaluation and policy improvement in a single step. Unlike policy iteration, where the policy is evaluated to convergence in each iteration, value iteration performs only a partial evaluation by using a single sweep of updates. This allows for interleaving partial evaluation with policy improvement, making it a popular Generalized Policy Iteration (GPI) method.

The value iteration algorithm repeatedly applies the Bellman optimality equation to iteratively refine the value function:

$$V_{k+1}(s) = \max_a \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s' \mid s, a) V_k(s') \right]$$

This recursive update simultaneously approximates the optimal value function $V^*(s)$ while implicitly improving the policy $\pi(s)$.

It can be shown that as the number of iterations $k \to \infty$, V_k tends to the optimal value function $V^*(s)$.

Once $V^*(s)$ is computed, the optimal policy π^* can be derived by acting greedily with respect to the optimal value function:

$$\pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s',r} \Pr(s', r \mid s, a)[r + \gamma V(s')]$$

4.3.2.1 Value iteration algorithm

The algorithm alternates between updating the value function and checking the magnitude of the updates to determine convergence.

Algorithm 3 Value iteration

```
1: Initialize $V(s)$ for all $s \in \mathcal{S}^+$ arbitrarily
2: $V(\text{terminal}) = 0$
3: repeat
4:     $\Delta = 0$
5:     for each $s \in \mathcal{S}$ do
6:         $v = V(s)$
7:         $V(s) = \max_a \sum_{s',r} \Pr(s', r \mid s, a)\,[r + \gamma V(s')]$
8:         $\Delta = \max(\Delta, |v - V(s)|)$
9:     end for
10: until $\Delta < \theta$
```

Value iteration is more computationally efficient than policy iteration because it skips the full policy evaluation step. The algorithm is guaranteed to converge to the optimal value function $V^*(s)$.

4.3.3 Asynchronous Dynamic Programming

Asynchronous Dynamic Programming represents a more flexible alternative to classical Dynamic Programming methods, which rely on exhaustive sweeps over the entire state space. Instead of systematically updating all states in each iteration, Asynchronous Dynamic Programming applies backups selectively to individual states, allowing for potentially faster convergence and scalability. Despite its asynchronicity, this method is still guaranteed to converge to the optimal value function $V^*(s)$ and policy $\pi^*(s)$.

Asynchronous Dynamic Programming extends the applicability of Dynamic Programming to larger and more complex problems by avoiding exhaustive state sweeps and leveraging selective updates. While it alleviates some of the computational burden, the curse of dimensionality remains a significant challenge, requiring additional strategies such as approximation methods or alternative frameworks to tackle extremely large-scale Markov Decision Processes.

When Dynamic Programming methods become impractical due to the size of the state space, alternative techniques like Linear Programming can be used to solve Markov Decision Processes. However, Linear Programming scales poorly for very large problems, making it unsuitable for high-dimensional or massive state spaces where Dynamic Programming methods like Asynchronous Dynamic Programming might still perform better.

4.4 Reinforcement Learning

On-policy learning refers to an approach where the agent learns value functions based on the same policy it uses to select actions. One of the key challenges of on-policy learning is finding the right balance between exploration (trying new things) and exploitation (sticking with what works). This can make it difficult for the agent to converge on an optimal deterministic policy.

In contrast, off-policy learning allows the agent to use a different behavior policy, $b(a \mid s)$, to select actions while learning the value functions of a target policy, $\pi(a \mid s)$. This approach offers more flexibility because the agent can explore using a behavior policy, while still learning towards an optimal deterministic policy, $\pi^*(a \mid s)$.

However, regardless of the behavior policy used, it's impossible to learn a policy $\pi(a \mid s)$ if there are actions in that state with zero probability according to the behavior policy $b(a \mid s)$. This situation arises when the behavior policy never transitions to a particular state from the current one, making it impossible to learn about that state.

Importance sampling Importance sampling is a technique that allows us to estimate expectations of a distribution that differs from the one used to generate the samples. There are two main types of sampling methods for importance sampling:

- *Ordinary*: this method is unbiased but has higher variance
$$V_\pi(s) \approx \frac{\sum_i \rho[i]\text{Return}[i]}{N(s)}$$
- *Weighted*: this method is biased (though the bias decreases over time) and has lower variance:
$$V_\pi(s) \approx \frac{\sum_i \rho[i]\text{Return}[i]}{\sum_i \rho[i]}$$

4.4.1 Monte Carlo

Monte Carlo methods are a class of Reinforcement Learning techniques that rely purely on experience (data) to learn value functions and policies. These methods do not require a model or simulation of the environment and learn from the complete returns of episodes. They are specifically designed for episodic tasks, where each episode has a clear beginning and end.

4.4.1.1 Policy iteration

In Monte Carlo policy evaluation, the objective is to learn the state value function $V_\pi(s)$ by using episodes generated from a policy π. We do this by averaging the returns observed after visiting each state s during the episode.

Monte Carlo policy evaluation can be implemented in two ways:

- *Every visit Monte Carlo*: average returns for every visit to state s during an episode.
- *First visit Monte Carlo*: average returns only for the first visit to state s in an episode.

Both methods converge to the correct value asymptotically, estimating the value of each state based on the average return from multiple visits. The update rule for Monte Carlo policy evaluation is:
$$V(s_t) = V(s_t) + \alpha(v_t - V(s_t))$$

Here, $v_t = \sum_{l=t}^{T} \gamma^{l-t} r_{l+1}$ is the discounted sum of rewards, and $\alpha > 0$ is the learning rate.

Algorithm 4 Monte Carlo policy evaluation

```
1: Initialize $V(s) \in \mathbb{R}$ arbitrarily, for all $s \in \mathcal{S}$          ▷ Initialization
2: Initialize Returns($s$) as an empty list, for all $s \in \mathcal{S}$
3: repeat
4:     for each episode do
5:         Generate an episode following $\pi : S_0, A_0, R_1 S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$
6:         $G = 0$
7:         for each step of episode $t = T-1, T-2, \dots, 0$ do
8:             $G = \gamma G + R_{t+1}$
9:             if $S_t \notin \{S_0, S_1 \dots, S_{t-1}\}$ then:
10:                Append $G$ to Returns($S_t$)
11:                $V(S_t)$ = average(Returns($S_t$))
12:            end if
13:        end for
14:    end for
15: until true
```

To improve the policy, we aim to find a policy that maximizes the action-value function $Q_\pi(s, a)$. The policy can be improved by selecting the action a that maximizes $Q_\pi(s, a)$ for each state s:

$$\pi'(s) = \operatorname*{argmax}_a Q_\pi(s, a)$$

To achieve this, we average the return starting from a state-action pair and following the policy π. This method also converges asymptotically if every state-action pair is visited.

To ensure full exploration of the state-action space, exploring starts can be used. In this approach, both the first state and action are chosen randomly, ensuring that all state-action pairs have a nonzero probability of being explored.

Algorithm 5 Monte Carlo policy iteration

```
1: $\pi(s) \in \mathcal{A}(s)$ arbitrarily, for all $s \in \mathcal{S}$
2: $Q(s, a) \in \mathbb{R}$ arbitrarily, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
3: Returns($s, a$) = empty list, for all $s2\mathcal{S}$, $a \in \mathcal{A}(s)$
4: loop
5:     Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability greater than
   zero
6:     Generate an episode from $S_0$, $A_0$, following $\pi : S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$
7:     $G = 0$
8:     for each step of episode, $t = T_1, T_2, \dots, 0$ do
9:         $G = \gamma G + R_{t+1}$
10:        if $S_t, A_t \notin S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$ then
11:            Append $G$ to Returns($S_t, A_t$)
12:            $Q(S_t, A_t)$ = average(Returns($S_t, A_t$))
13:            $\pi(S_t) = \operatorname{argmax}_a Q(S_t, a)$
14:        end if
15:    end for
16: end loop
```

4.4.1.2 Epsilon-greedy Monte Carlo policy iteration

Exploring starts is a simple and effective strategy, but it is not always practical. To continue exploration throughout the learning process, we introduce the exploration-exploitation dilemma.

One of the simplest solutions to this dilemma is the ε-greedy exploration strategy. Instead of directly optimizing the deterministic policy, we seek an optimal ε-soft policy:

$$\pi(a \mid s) = \begin{cases} \frac{\varepsilon}{|\mathcal{A}(s)|} + 1 - \varepsilon & \text{if } a^* = \operatorname{argmax}_{a \in \mathcal{A}} Q_\pi(s,a) \\ \frac{\varepsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases}$$

Algorithm 6 ε-soft Monte Carlo policy iteration

```
1: $\pi$=an arbitrary $\varepsilon$-soft policy
2: $Q(s,a) \in \mathbb{R}$ arbitrarily, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
3: Returns$(s,a)$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
4: loop for each episode
5:     Generate an episode following $\pi : S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
6:     $G = 0$
7:     for each step of episode, $t = T-1, T-2, \ldots, 0$ do
8:         $G = \gamma G + R_{t+1}$
9:         if $S_t, A_t \notin S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$ then
10:            Append $G$ to Returns$(S_t, A_t)$
11:            $Q(S_t, A_t)$ =average(Returns$(S_t, A_t)$)
12:            $A^* = \operatorname{argmax}_a Q(S_t, a)$        ▷ Ties broken arbitrarily
13:            for $a \in \mathcal{A}(S_t)$ do
14:                $\pi(a \mid S_t) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(S_t)|} & \text{if } a = A^* \\ \frac{\varepsilon}{|\mathcal{A}(S_t)|} & \text{if } a \neq A^* \end{cases}$
15:            end for
16:        end if
17:    end for
18: end loop
```

Theorem 4.4.1. *Any ε-greedy policy π' with respect to Q_π is an improvement over any ε-soft policy π.*

4.4.2 Temporal difference learning

Temporal Difference learning is a powerful method for updating state values through bootstrapping, which means that it updates the value of a state using the current estimate of the value for the next state. This approach combines ideas from both Monte Carlo methods and Dynamic Programming, giving it a strong advantage in terms of flexibility and efficiency. The Temporal Difference update rule is as follows:

$$V(s_t) = V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$$

Algorithm 7 Temporal difference policy evaluation

```
 1: Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}^+$
 2: $V(\text{terminal}) = 0$
 3: for each episode do
 4:     Initialize $S$
 5:     repeat for each step of episode
 6:         $A =$ action given by $\pi$ for $S$
 7:         Take action $A$, observe $R$, $S'$
 8:         $V(S) = V(S) + \alpha[R + \gamma V(S') - V(S)]$
 9:         $S = S'$
10:     until $S$ is terminal
11: end for
```

4.4.2.1 SARSA

SARSA is an on-policy Reinforcement Learning algorithm used for evaluating and improving a policy. As an on-policy method, it means that the policy used to make decisions is the same policy that is being improved.

The SARSA algorithm updates the action-value function, $Q(S_t, A_t)$, based on the observed reward and the estimated value of the next state-action pair, $Q(S_{t+1}, A_{t+1})$. The update rule for SARSA is given by:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

SARSA is often combined with an ε-greedy policy to balance exploration and exploitation. In this policy, the agent mostly chooses the action with the highest estimated value, but occasionally explores other actions with probability ε, ensuring that the agent continues to explore the state space.

The key parameters of the SARSA algorithm are the learning rate $\alpha \in (0, 1]$ and the exploration rate $\varepsilon > 0$, which controls the amount of exploration in the policy.

Algorithm 8 SARSA

```
 1: Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$
 2: $Q(\text{terminal}, \cdot) = 0$
 3: loop
 4:     Initialize $S$
 5:     Choose $A$ from $S$ using policy derived from $Q$
 6:     repeat for each step of episode
 7:         Take action $A$, observe $R$, $S'$
 8:         Choose $A'$ from $S'$ using policy derived from $Q$
 9:         $Q(S, A) = Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$
10:         $S = S'$
11:         $A = A'$
12:     until $S$ is terminal
13: end loop
```

4.4.2.2 Q-Learning

Q-Learning is a popular off-policy algorithm in Reinforcement Learning used to estimate the optimal action-value function $Q^*(s, a)$. Unlike on-policy methods like SARSA, which evaluates and improves the same policy used for action selection, Q-Learning is off-policy, meaning it evaluates the optimal policy while potentially following a different policy for action selection.

Q-Learning is based on the Bellman optimality equation, rather than the Bellman expectation equation used in SARSA. The update rule for Q-Learning is given by:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right)$$

The key difference between Q-Learning and SARSA is that, while SARSA updates $Q(S_t, A_t)$ using the next action A_{t+1} chosen by the policy, Q-Learning updates $Q(S_t, A_t)$ using the maximum action value in the next state, effectively trying to find the optimal policy. Q-Learning is often used with an ε-greedy policy, where the agent mostly selects actions based on the highest action value, but occasionally explores other actions with a small probability ε.

Algorithm 9 Q-Learning

```
1: Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$
2: $Q(\text{terminal}, \cdot) = 0$
3: loop
4:     Initialize $S$
5:     repeat for each step of episode
6:         Choose $A$ from $S$ using policy derived from $Q$
7:         Take action $A$, observe $R$, $S'$
8:         $Q(S, A) = Q(S, A) + \alpha \left(R + \gamma \max_a Q(S', a) - Q(S, A)\right)$
9:         $S = S'$
10:    until $S$ is terminal
11: end loop
```

In this algorithm, the agent interacts with the environment, selects actions based on the policy derived from the current estimates of Q, and updates the action-value function based on the observed rewards and the maximum possible value of the next state. This process continues until the agent converges to the optimal policy.

4.4.2.3 Eligibility traces

Eligibility traces are an important concept in Reinforcement Learning that enhance the efficiency of updating value estimates for states or actions. They are primarily used in conjunction with Temporal Difference learning methods, such as SARSA and Q-Learning. Eligibility traces enable faster and more efficient learning by assigning credit to past experiences based on their contribution to future rewards. The key features of eligibility traces are:

1. *Temporal credit assignment*: eligibility traces allow RL algorithms to assign credit or blame to actions taken in the past for the rewards received later.

2. *Memory mechanism*: eligibility traces act as a form of memory, maintaining a record of recently visited state-action pairs.

3. *Decay factor*: the decay factor controls how much influence past experiences have on the current update.

4. *Updating value estimates*: when a reward is received, eligibility traces guide the updating of value estimates for relevant states or actions.

5. *Efficiency and learning speed*: eligibility traces can accelerate the learning process by allowing updates to propagate through the environment more efficiently.

4.4.3 Monte Carlo and Temporal Difference

Temporal Difference learning is more flexible than Monte Carlo methods, particularly when dealing with incomplete sequences or continuous tasks. While Monte Carlo has the advantage of providing unbiased estimates, Temporal Difference's bootstrapping mechanism generally leads to lower variance and faster learning. However, Temporal Difference methods can be more sensitive to initial values and may struggle with function approximation compared to Monte Carlo. Both methods, however, share the key benefit of being model-free, meaning they do not require knowledge of the environment's dynamics to update their value functions.

	Monte Carlo	**Temporal Difference**
Update	End of an episode	After each step within an episode
Episodes	Complete	Partial
Task	Episodic	Episodic and continuous
Estimation bias	No	Yes
Estimation variance	Higher variance	Lower variance
Initialization	Less sensitive	More sensitive
Model dependency	Model-free	Model-free
Update method	Sampling	Bootstrapping

4.5 Multi-Armed Bandits

The Multi-Armed Bandit problem is a simplified framework of Reinforcement Learning, often viewed as a special case of a Markov Decision Process. It can be described as follows:

- *State space* ($\mathcal{S}$): a single state, representing the lack of contextual information.
- *Action space* ($\mathcal{A}$): a set of N actions, or arms.
- *Transition probabilities* (P): state transitions are trivial, and the probability of remaining in the same state is always 1.
- *Reward function* (R): rewards depend solely on the action
- *Discount factor* (γ): fixed to 1.
- *Initial probabilities* (μ_0): fixed to 1.

The only component left to define is the reward structure, which can vary depending on the nature of the problem:

- *Deterministic*: each arm yields a fixed reward. This is trivial to solve.

- *Stochastic*: rewards are drawn from a stationary probability distribution.
- *Adversarial*: rewards are chosen strategically by an adversary who knows the agent's algorithm.

In essence, the k-Armed Bandit problem requires an agent to select from k actions and receive rewards based on those choices, aiming to maximize the total reward over time. Unlike typical Markov Decision Processes, the Multi-Armed Bandit setting involves decisions made in isolation, without a broader state context.

4.5.1 Expexted regret

The agent interacts with the environment as follows:

- At each round t, the agent selects an arm a_{i_t}.
- The environment generates a reward $r_{a_{i_t},t}$ from the distribution $\mathcal{R}(a_{i_t})$.
- The agent updates its knowledge based on the history h_t (previous actions and rewards).

The optimal arm a^* has the highest expected reward:

$$R^* = R(a^*) = \max_{a \in \mathcal{A}} R(a) = \max_{a \in \mathcal{A}} \mathbb{E}_{r \sim \mathcal{R}(a)}[r]$$

When the agent selects arm a_{i_t}, it incurs an instantaneous regret:

$$\Delta_{i_t} = R^* - R(a_{i_t})$$

Here, Δ_i is known as the suboptimality gap for arm a_i. The goal is to minimize the expected cumulative regret over T rounds:

$$L_T = TR^* - \mathbb{E}\left[\sum_{t=1}^{T} R(a_{i_t})\right]$$

Minimizing cumulative regret is equivalent to maximizing cumulative reward. A good algorithm satisfies the no-regret property, where:

$$\frac{L_T}{T} \to 0 \text{ as } T \to \infty$$

The difficulty of the problem depends on the similarity of the arms: the closer their rewards, the harder it is to identify the best one.

Theorem 4.5.1. *For stochastic Multi-Armed Bandit problems, any algorithm satisfies the regret bound:*

$$L_T \geq \log T \sum_{a_i \in \mathcal{A}: \Delta_i > 0} \frac{\Delta_i}{KL(\mathcal{R}(a_i), \mathcal{R}(a^*))}$$

as $T \to \infty$, where $KL(\mathcal{R}(a_i), \mathcal{R}(a^*))$ is the Kullback-Leibler divergence between the reward distributions of arm a_i and the optimal arm a^*. High regret arises when the rewards of different arms are very similar, making it challenging to distinguish the best arm.

Action values In bandit problems, the agent uses action-value estimates Q_n to guide decision-making. The update rule depends on whether the problem is stationary or non-stationary:

- *Non-stationary problems*:
$$Q_{n+1} = Q_n + \alpha\,(R_n - Q_n)$$
Here, α is a step-size parameter that adjusts over time.

- *Stationary problems*:
$$Q_{n+1} = Q_n + \frac{1}{n}\,(R_n - Q_n)$$
Here, α is typically set to $\frac{1}{n}$, the reciprocal of the number of times the action has been taken.

Initialization Action-value estimates are traditionally initialized to zero, but alternative initialization strategies can influence exploration. In optimistic initialization we set high initial values encourages exploration by making all arms initially appealing. However, this approach is less effective in non-stationary problems, where the environment evolves over time.

The challenge is choosing appropriate optimistic values is non-trivial, as they must balance exploration and exploitation effectively.

4.5.2 Action selection

Selecting the action with the highest value is not always optimal, as it may fail to balance immediate rewards with long-term benefits. Therefore, it is essential to strike a balance between exploration and exploitation:

- *Exploitation*: the agent uses its current knowledge to maximize immediate rewards.
- *Exploration*: the agent seeks to gather additional information to improve decision-making for future rewards.

The ε-greedy approach balances exploration and exploitation by performing the greedy action most of the time but occasionally exploring other options. Mathematically:

$$A_t = \begin{cases} \text{argmax}_a\, Q_t(a) & \text{with probability } 1-\varepsilon \\ \text{Uniform}(\{a_1, \dots, a_k\}) & \text{with probability } \varepsilon \end{cases}$$

Two main formulations exist for Multi-Armed Bandit algorithms:

- *Frequentist*: the rewards $R(a_1), \dots, R(a_N)$ are unknown parameters. A policy selects an arm at each time step based on the observation history.
- *Bayesian*: the rewards $R(a_1), \dots, R(a_N)$ are treated as random variables with prior distributions $f_1, \dots, f_N$. The policy selects an arm based on the observation history and the prior information.

Upper Confidence Bound (UCB) The UCB algorithm is a frequentist approach to the stochastic Multi-Armed Bandit problem. Instead of relying solely on empirical estimates, it computes an upper bound $U(a_i)$ for the expected reward $R(a_i)$ with high probability:

$$U(a_i) = \hat{R}_t(a_i) + B_t(a_i)$$

where $B_t(a_i)$ represents the uncertainty in the estimate $\hat{R}_t(a_i)$, depending on the number of times arm a_i has been pulled, $N_t(a_i)$. Specifically:

- Small $N_t(a_i)$, encouraging exploration.
- Large $N_t(a_i)$, favoring exploitation.

The selected action at each time step is:

$$A_t = \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln(t)}{N_t(a)}} \right]$$

Here, $Q_t(a)$ represents the exploitation term, c is a user-defined coefficient, and $\sqrt{\frac{\ln(t)}{N_t(a)}}$ accounts for exploration.

Theorem 4.5.2. *At a finite time horizon T, the expected cumulative regret of the UCB1 algorithm for a stochastic Multi-Armed Bandit problem satisfies:*

$$L_T \leq 8 \log T \sum_{a_i \in \mathcal{A}:\Delta_i > 0} \frac{1}{\Delta_i} + \left(1 + \frac{\pi^2}{3}\right) \sum_{a_i \in \mathcal{A}:\Delta_i > 0} \Delta_i$$

Thompson sampling Thompson sampling is a Bayesian approach to stochastic Multi-Armed Bandit problems. It uses Bayesian priors for each arm's reward distribution $f_1, \ldots, f_N$ and updates these distributions based on observed outcomes:

- At each round t, sample a reward estimate $\hat{r}_1, \ldots, \hat{r}_N$ from the current posterior distributions.
- Select the arm with the highest sampled value:

$$a_{i_t} = \underset{i}{\operatorname{argmax}}\, \hat{r}_i$$

- Update the prior distribution for the selected arm based on the observed reward.

Theorem 4.5.3. *At time T, the expected cumulative regret of Thompson sampling for a stochastic Multi-Armed Bandit problem satisfies:*

$$L_T \leq \mathcal{O}\left(\sum_{a_i \in \mathcal{A}:\Delta_i > 0} \frac{\Delta_i}{KL(\mathcal{R}(a_i), \mathcal{R}(a^*))} (\log T + \log \log T) \right)$$

Here, $KL(\mathcal{R}(a_i), \mathcal{R}(a^*))$ is the Kullback-Leibler divergence between the reward distributions of arm a_i and the optimal arm a^*.