Machine Learning

Theory

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 Chernoff/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 Radial Basis Networks, Gaussian Processes, and Support Vector Machines. Additionally, topics like model selection, feature selection, ensemble methods (e.g., bagging and boosting), 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 (MDPs), Bellman equations, value iteration, policy iteration, TD, SARSA, Q-learning, value function approximation, policy search, REINFORCE, POMDPs, and the Multi-Armed Bandit problem.

Contents

CHAPTER 1

Introduction

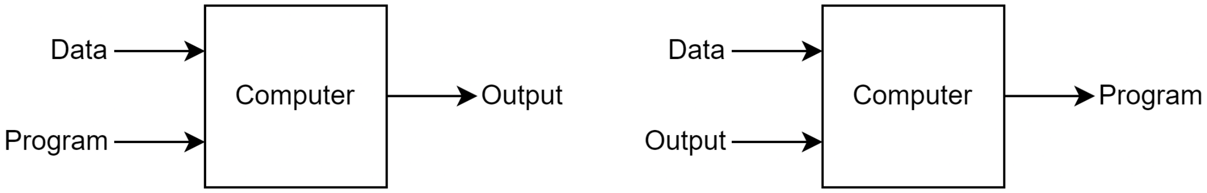
1.1 Machine Learning

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 MDP*: an extension of MDP 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 $\Pr(t|x)$ and then marginalizing to find the conditional mean:
$$\mathbb{E}\left[t|x\right] = \int t \Pr(t|x)\, dt$$
- *Discriminative*: this method models the joint density $\Pr(x,t)$, infers the conditional density $\Pr(t|x)$, and then marginalizes to find the conditional mean:
$$\mathbb{E}\left[t|x\right] = \int t \Pr(t|x)\, dt$$

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:

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

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

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

  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:

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

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

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

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:

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

This solution can be easily decoupled for each output k:

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

An advantage of this approach is that $\left(\mathbf{\Phi}^T \mathbf{\Phi}\right)^{-1}$ 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	$\phi_j(x) = \dfrac{1}{1+e^{\frac{\mu_j - x}{\sigma}}}$

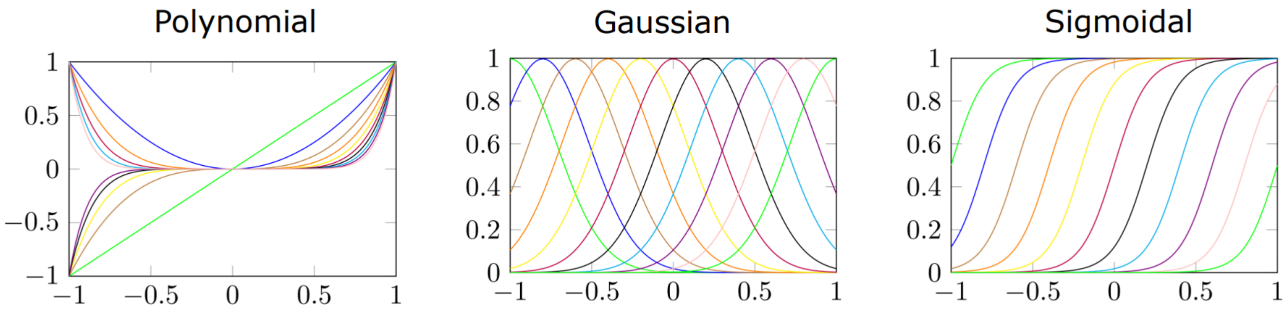


Figure 2.1: Polynomial, Gaussian, and sigmoidal basis functions

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

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

$$L(\mathbf{w}) = L_D(\mathbf{w}) + \lambda L_W(\mathbf{w})$$

Here, $L_D(\mathbf{w})$ represents the usual loss function, $L_W(\mathbf{w})$ 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:

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

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

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

  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:

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

  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:

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

  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:

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

  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:

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

   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 the F-statistic 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 $\Pr(\mathcal{D}|\mathbf{w})$.

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:

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

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:

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

Here, $\mathcal{N}(t_n|\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_n), \sigma^2)$ 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:

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

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 $\ell(\mathbf{w})$ with respect to $\mathbf{w}$ to zero:

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

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 (OLS) 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:

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

   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:

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

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

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

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:

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

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:

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

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:

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

  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:

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

  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:

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

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 $k = 1, \ldots, K$) 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 $\Pr(C_k|\mathbf{x})$ and learning the model parameters from the data.
- *Probabilistic generative approach*: modeling $\Pr(\mathbf{x}|C_k)$ and class priors $\Pr(C_k)$, fitting models to the data, and inferring the posterior using Bayes' rule.

2.3.1 Discriminant function

In linear classification, we will use generalized linear models:

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

Here, the function $f(\cdot)$ is not linear in $\mathbf{w}$ due to the presence of the non-linear activation function f, which yields either a discrete label or a probability value as its output. The function $f(\cdot)$ partitions the input space into decision regions, with their boundaries known as decision boundaries or decision surfaces. Notably, these decision surfaces 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.

> **Example:**
> For instance, in a problem with $K = 5$ classes, a data sample belonging to class 4 would be encoded as:
> $$t = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix}^T$$

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 $\frac{y(\mathbf{x})}{\|\mathbf{w}\|_2}$.

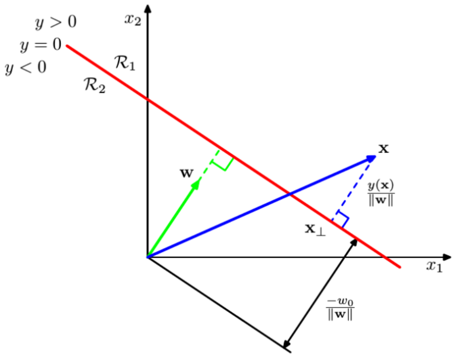


Figure 2.2: Two-class decision problem boundaries

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 (C_i) 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 C_i and C_j. Similar to the one versus the rest approach, this method also suffers from ambiguity.

One potential 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.

Linear basis function models 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:

$$\mathbf{y}(\mathbf{x}) = \tilde{\mathbf{W}}^T \tilde{\mathbf{x}}$$

Here, $\tilde{\mathbf{W}}$ is of size $(D+1) \times K$, where its k-th column is denoted by $\tilde{\mathbf{w}}_k = \left(w_{k0}, \mathbf{w}_k^T\right)^T$, and $\tilde{\mathbf{x}} = \left(1, \mathbf{x}^T\right)^T$.

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:

$$\tilde{\mathbf{W}} = \left(\tilde{\mathbf{X}}^T \tilde{\mathbf{X}}\right) \tilde{\mathbf{X}}^T \tilde{\mathbf{T}}$$

Here, $\tilde{\mathbf{X}}$ is an $N \times (D+1)$ matrix with its i-th row being $\tilde{\mathbf{x}}_i^T$ and $\mathbf{T}$ is an $N \times K$ matrix with its i-th row as $\mathbf{t}_i^T$. In this setup, any new sample $\tilde{\mathbf{x}}_{new}^T$ is assigned to class C_k if $t_k > t_j$ for all j, where t_k represents the k-th component of the model output computed as $t_k = \tilde{\mathbf{x}}^T \tilde{\mathbf{w}}_k$.

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.

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 linear discriminant model designed specifically for two-class problems, with class encoding as $\{-1, 1\}$. 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. Therefore, stochastic gradient descent is commonly employed for optimization in perceptron learning.

The core concept of the perceptron is to optimize $\mathbf{w}$ such that $\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i) \geq 0$ for $\mathbf{x}_i \in C_1$ and $\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i) < 0$ otherwise. The perceptron criterion is expressed as:

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

Here, correctly classified samples do not contribute to L, and each misclassified sample $\mathbf{x}_i \in \mathcal{M}$ contributes as $\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_n) t_n$.

Minimizing L_P is achieved using stochastic gradient descent:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha \nabla L_P(\mathbf{w}) = \mathbf{w}^{(k)} + \alpha \boldsymbol{\phi}(\mathbf{x}_n) t_n$$

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 algorithm

```
Initialize $\mathbf{w}_0$
$k \leftarrow 0$
repeat
    $k \leftarrow k + 1$
    $n \leftarrow k \mod N$
    if $\hat{t}_n \neq t_n$ then
        $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \boldsymbol{\phi}(\mathbf{X}_n) t_n$
    end if
until convergence
```

Theorem 2.3.1 (Perceptron convergence)**.** *If the training dataset is linearly separable in the feature space $\boldsymbol{\Phi}$, 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 parameter initialization and the order of updates.

2.3.2 Probabilistic discriminative approaches

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

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

Here, $\sigma(\cdot)$ denotes the sigmoidal function. This model is commonly referred to as logistic regression.

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, i.e., the probability of observing the targets given the inputs $\Pr(\mathbf{t} | \mathbf{X}, \mathbf{w})$. We model the likelihood of a single sample using a Bernoulli distribution, employing the logistic regression model for conditioned class probability:

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

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

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

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

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

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

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

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 (even in an online learning setting).

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

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

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

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

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

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

Then, we compute the gradient for each weight vector:

$$\nabla L_{\mathbf{w}_j}(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{n=1}^{N} (y_{nj} - t_{nj})\,\boldsymbol{\phi}_n$$

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

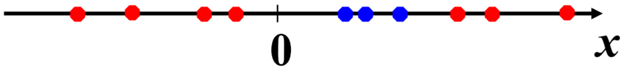
2.4 Kernel methods

Frequently, we seek to detect nonlinear patterns within our datasets. In nonlinear regression, the connection between input and output may deviate from linearity, while in nonlinear classification, class boundaries might not be linearly separable. Linear models often prove insufficient in capturing such complexities. However, kernel methods offer a solution by transforming data into higher-dimensional spaces where linear relationships become apparent, thereby enabling linear models to effectively operate in nonlinear scenarios.

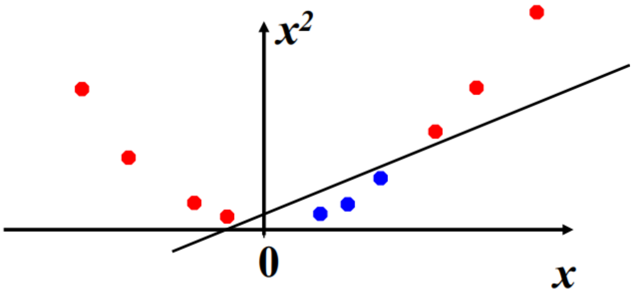
The process of transforming the original input space into a feature space is termed feature mapping, denoted as:

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

Example:
Consider a binary classification problem where no linear separator exists:



Now, let's map the input space (a single variable x) to a feature space with two features: $x \to \{x, x^2\}$ As a result, the data becomes linearly separable:



This concept extends naturally to higher dimensions and more intricate problem domains.

However, a significant drawback arises known as the curse of dimensionality. This occurs due to the exponential growth in the number of features as the input variables increase, rendering the mapping computationally infeasible. Kernel methods offer a solution to this challenge by bypassing the need for explicit computation of the feature mapping. While they are computationally intensive, they remain feasible for practical implementation.

2.4.1 Kernel function

The kernel function is defined as the scalar product between the feature vectors of two data samples:

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

The kernel function exhibits symmetry: $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$. It can be interpreted as a measure of similarity between $\mathbf{x}$ and $\mathbf{x}'$.

Interestingly, very large feature vectors, even infinite ones, can result in a kernel function that is computationally tractable.

Certain special classes of kernel functions exist:

- *Stationary kernels*: $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$.
- *Homogeneous kernels* (or radial basis functions): $k(\mathbf{x}, \mathbf{x}') = k(\|\mathbf{x} - \mathbf{x}'\|)$.

Kernel function design We are not obligated to compute the kernel function by first generating the feature space, as we aim to avoid explicitly calculating the feature vectors. Two primary approaches exist for designing a kernel function:

- Create kernel functions directly from scratch.
- Design kernel functions by applying a predefined set of rules to existing ones.

In both cases, it's crucial to ensure that the resulting kernel functions are valid, meaning they correspond to a scalar product in some feature space.

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

For this theorem, the necessary and sufficient condition for a function $k(\mathbf{x}, \mathbf{x}')$ to be a valid kernel is that the Gram matrix $\mathbf{K}$ is positive semi-definite for all possible choices of $\mathcal{D} = \{x_i\}$. This condition implies that $\mathbf{x}^T\mathbf{K}\mathbf{x} > 0$ for any non-zero real vector $\mathbf{x}$, meaning that the double sum $\sum_i \sum_j \mathbf{K}_{ij}\mathbf{x}_i\mathbf{x}_j$ is strictly positive for any real numbers $\mathbf{x}_i$ and $\mathbf{x}_j$.

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:

1. $k_(\mathbf{x}, \mathbf{x}') = ck_1(\mathbf{x}, \mathbf{x}')$, where $c > 0$ is a constant.
2. $k_(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})k_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}')$, where $f(\cdot)$ is any function.
3. $k_(\mathbf{x}, \mathbf{x}') = q\left(k_1(\mathbf{x}, \mathbf{x}')\right)$, where $q(\cdot)$ is a polynomial with non-negative coefficients.
4. $k_(\mathbf{x}, \mathbf{x}') = e^{k_1(\mathbf{x}, \mathbf{x}')}$.
5. $k_(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}')$.
6. $k_(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}')k_2(\mathbf{x}, \mathbf{x}')$.
7. $k_(\mathbf{x}, \mathbf{x}') = k_3(\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}'))$, where $\boldsymbol{\phi}(\mathbf{x})$ maps $\mathbf{x}$ to $\mathbb{R}^M$ and $k_3(\cdot, \cdot)$ is a valid kernel in $\mathbb{R}^M$.
8. $k_(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is a symmetric semidefinite matrix.
9. $k_(\mathbf{x}, \mathbf{x}') = k_a(\mathbf{x}_a, \mathbf{x}_a') + k_b(\mathbf{x}_b, \mathbf{x}_b')$.
10. $k_(\mathbf{x}, \mathbf{x}') = k_a(\mathbf{x}_a, \mathbf{x}_a')k_b(\mathbf{x}_b, \mathbf{x}_b')$.

Kernel trick It's feasible to modify the representation of linear models by substituting terms involving $\phi(\mathbf{x})$ with alternatives solely based on $k(\mathbf{x}, \cdot)$. In essence, the output of a linear model can be computed solely based on the similarities between data samples, as computed with the kernel function.

This methodology, known as the kernel trick, finds application in various learning algorithms including: ridge regression, $K-NN$ regression, perceptron, nonlinear PCA, and support vector machines.

Gaussian kernel The Gaussian kernel is a widely employed kernel function in various Machine Learning algorithms. Its mathematical representation is given by:

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

This kernel function defines a similarity measure between two vectors $\mathbf{x}$ and $\mathbf{x}'$ in the feature space. It assigns higher similarity to vectors that are closer to each other, based on the Euclidean distance, with σ controlling the width of the kernel.

Additionally, the Gaussian kernel can be generalized by replacing the dot product $\mathbf{x}^T\mathbf{x}'$ with a nonlinear kernel function $\kappa(\mathbf{x}, \mathbf{x}')$. This leads to the extended form of the Gaussian kernel:

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

This extension allows the Gaussian kernel to operate in a more flexible feature space, potentially capturing nonlinear relationships between data points, thereby enhancing its applicability in various Machine Learning tasks.

Symbolic data kernel Kernel methods are not limited to real vectors as inputs; they can be extended to various data structures such as graphs, sets, strings, texts, and more. The kernel function serves as a measure of similarity between two samples. For example, in the case of sets, a common kernel function is employed:

$$k(A_1, A_2) = 2^{|A_1 \cap A_2|}$$

This kernel function quantifies the similarity between two sets A_1 and A_2 by computing the cardinality of their intersection. The resulting value reflects the degree of overlap between the sets, indicating their similarity.

Generative model kernel Kernel functions can also be defined using probability distributions. In the context of generative models, where $\mathrm{P}(\mathbf{x})$ represents the probability distribution, a kernel function can be defined as:

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

This kernel function is valid as it corresponds to the inner product in a one-dimensional feature space obtained by mapping $\mathbf{x}$ to $\mathrm{P}(\mathbf{x})$. It effectively measures the similarity between two samples by considering their respective probabilities under the generative model.

2.4.2 Kernel ridge regression

The loss function utilized in ridge regression is given by:

$$L(\mathbf{w}) = \frac{1}{2}(\mathbf{t} - \mathbf{\Phi}\mathbf{w})^T (\mathbf{t} - \mathbf{\Phi}\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

To solve it, we equate the gradient of L with respect to $\mathbf{w}$ to zero:

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \lambda\mathbf{w} - \mathbf{\Phi}^T(\mathbf{t} - \mathbf{\Phi}\mathbf{w}) = 0$$

Now, instead of solving it for $\mathbf{w}$, let's perform a variable change ($\mathbf{a} = \lambda^{-1}(\mathbf{t} - \mathbf{\Phi}\mathbf{w})$):

$$\mathbf{w} = \mathbf{\Phi}^T\lambda^{-1}(\mathbf{t} - \mathbf{\Phi}\mathbf{w}) = \mathbf{\Phi}^T\mathbf{a}$$

Substituting $\mathbf{w}$ in the gradient, we have:

$$\begin{aligned}
&\lambda\mathbf{w} - \mathbf{\Phi}^T(\mathbf{t} - \mathbf{\Phi}\mathbf{w}) = 0 \rightarrow \\
&\mathbf{\Phi}^T\left(\lambda\mathbf{a} - \left(\mathbf{t} - \mathbf{\Phi}\mathbf{\Phi}^T\mathbf{a}\right)\right) = 0 \rightarrow \\
&\mathbf{\Phi}\mathbf{\Phi}^T\mathbf{a} + \lambda\mathbf{a} = \mathbf{t} \rightarrow \\
&\mathbf{a} = (\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{t}
\end{aligned}$$

Here, $\mathbf{K} = \mathbf{\Phi}\mathbf{\Phi}^T$ is known as the Gram matrix. The Gram matrix is an $N \times N$ matrix where each element represents the inner product between the feature vectors:

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \cdots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$$

The Gram matrix signifies the similarities between each pair of samples in the training data.

Prediction function To compute the prediction using the dual representation, we can utilize the following formula:

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

Here,$\mathbf{k}(\mathbf{x})$ is defined such that $k_n(\mathbf{x}) = k(\mathbf{x}_n, \mathbf{x})$ for all $\mathbf{x}_n \in \mathcal{D}$. Accordingly, the prediction is computed as the linear combination of the target values of the samples in the training set.

Comparison The original representation:

- Involves computing the inverse of $(\boldsymbol{\Phi}\boldsymbol{\Phi}^T + \lambda \mathbf{I}_M)$, which yields an $M \times M$ matrix.
- Is computationally convenient when M is relatively small.

The dual representation:

- Requires computing the inverse of $(\mathbf{K} + \lambda \mathbf{I}_N)$, which results in an $N \times N$ matrix.
- Is computationally favorable when N is very large or even infinite.
- Eliminates the need to explicitly compute $\boldsymbol{\Phi}$, enabling application to diverse data types such as graphs, sets, strings, and text.
- The computation of the similarity between data samples (i.e., the kernel function) is typically more efficient and simpler than calculating $\boldsymbol{\Phi}$.

2.4.3 Kernel regression

The k-nearest neighbors algorithm can be utilized for regression tasks by computing the average of the target values of the k nearest samples in the training data. This can be expressed as:

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

Nadaraya-Watson model In k-NN regression, the model output often exhibits significant noise due to the discontinuity of neighborhood averages. The Nadaraya-Watson model, also known as kernel regression, addresses this issue by employing a kernel function to calculate a weighted average of samples:

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

Typically, kernels are chosen based on their properties. Two common choices for kernels are:

- Epanechnikov Kernel (bounded support):

$$k(u) = \frac{3}{4}\big(1 - u^2\big) \qquad |u| \leq 1$$

- Gaussian Kernel (infinite support):

$$K(u) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{u^2}{2\sigma^2}}$$

2.4.4 Gaussian processes

Starting from the assumptions of Bayesian linear regression:

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

with the following prior probability:

$$\mathrm{P}(\mathbf{w}) = \mathcal{N}\left(\mathbf{w}|\mathbf{0}, \tau^2\mathbf{I}\right)$$

Now, let's compute the prior distribution of the outputs of the regression function:

$$\mathbf{y} = \mathbf{\Phi}\mathbf{w} \implies \mathcal{N}(\mathbf{y}|\boldsymbol{\mu}, \mathbf{S})$$

Here:

- $\boldsymbol{\mu} = \mathbb{E}[\boldsymbol{y}] = \mathbf{\Phi}\mathbb{E}[\boldsymbol{w}] = \mathbf{0}$
- $\mathbf{S} = \mathrm{Cov}(\mathbf{y}\mathbf{y}^T) = \mathbf{\Phi}\mathbb{E}[\boldsymbol{w}\boldsymbol{w}^T]\mathbf{\Phi}^T = \tau^2\mathbf{\Phi}\mathbf{\Phi}^T = \mathbf{K}$

In general, a Gaussian Process is defined as a probability distribution over a function $y(\mathbf{x})$ such that the set of values $y(\mathbf{x}_i)$ — for an arbitrary $\mathbf{x}_i$ — jointly have a Gaussian distribution. In our case:

$$\mathrm{P}(\mathbf{y}) = \mathcal{N}\left(\mathbf{y}|\mathbf{0}, \mathbf{K}\right)$$

where $\mathbf{K}$ is the Gram matrix defined as:

$$K_{nm} = k(\mathbf{x}_n, \mathbf{x}_m) = \tau^2\boldsymbol{\phi}(\mathbf{x}_n)^T\boldsymbol{\phi}(\mathbf{x}_m)$$

This provides a probabilistic interpretation of the Kernel function as:

$$k(\mathbf{x}_n, \mathbf{x}_m) = \mathbb{E}\left[y(\mathbf{x}_n), y(\mathbf{x}_m)\right]$$

We can apply the usual approaches to design the kernels. Two families of kernels typically used with Gaussian processes are:

- Gaussian kernel:
$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{\|\mathbf{x}-\mathbf{x}'\|_2^2}{2\sigma^2}}$$
- Exponential kernel:
$$k(\mathbf{x}, \mathbf{x}') = e^{-\theta|\mathbf{x}-\mathbf{x}'|}$$

2.5 Support Vector Machines

Kernel methods face a notable limitation: the need to compute the kernel function for every sample in the training set. Unfortunately, this computation can be computationally infeasible in practice. To address this challenge, sparse kernel methods seek solutions that rely only on a subset of the training samples. Two well-known sparse kernel methods are:

1. Support Vector Machines (SVMs).
2. Relevance Vector Machines.

2.5.1 Separable problems

The separation between data points can also be achieved using the perceptron algorithm. However, in this case, the final result is highly dependent on the initialization.

When choosing the best solution, consider the line that separates the points. Opt for the solution with fewer points close to that separating line. To address this, we can utilize the maximum margin classifier, which computes the margin as follows:

$$\text{margin} = \min_n \frac{t_n \left(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_n) + b\right)}{\|\mathbf{w}\|}$$

The goal is to find the optimal hyperplane by maximizing the expression:

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

However, solving this optimization problem can be very complex due to its computational demands and potential non-convexity.

To simplify the optimization problem, we first establish a canonical hyperplane across the separating variables. It's essential to acknowledge the existence of an infinite set of equivalent solutions represented by:

$$\kappa \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) + \kappa b \quad \forall \kappa > 0$$

However, we will focus solely on solutions that adhere to the condition:

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

Consequently, we transform the problem into an equivalent quadratic programming task aimed at minimizing:

$$\frac{1}{2} \|\mathbf{w}\|_2^2$$

subject to the constraint $t_n \left(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_n) + b\right) \geq 1$, for all n.

Dual problem We can obtain the dual problem by utilizing Lagrange multipliers, resulting in the following Lagrangian:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i (t_i(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i)) - 1)$$

To maximize $\mathcal{L}$ with respect to $\boldsymbol{\alpha}$ and minimize it with respect to $\mathbf{w}$ and b, we compute the gradients with respect to $\mathbf{w}$ and b and derive the dual representation:

$$\begin{cases} \frac{\partial}{\partial \mathbf{w}} \mathcal{L} = 0 \\ \frac{\partial}{\partial b} \mathcal{L} = 0 \end{cases} \rightarrow \begin{cases} \mathbf{w} = \sum_{i=1}^{n} \alpha_i t_i \boldsymbol{\phi}(\mathbf{x}_{\boldsymbol{i}}) \\ \sum_{i=1}^{n} \alpha_i t_i = 0 \end{cases}$$

This allows us to reformulate the optimization problem as the maximization of:

$$\tilde{\mathcal{L}}(\boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m)$$

subject to the constraints:

$$\begin{cases} \alpha_n \geq 0 \\ \sum_{n=1}^{N} \alpha_n t_n = 0 \qquad \forall n = 1, \dots, N \end{cases}$$

where the explicit feature mapping no longer appears explicitly.

Discriminant function The resulting discriminant function can be expressed as:

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

Here, only samples on the margin contribute, indicated by $\alpha_i > 0$. These crucial samples are known as the Support Vectors. The bias term, denoted as b, is computed as:

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

This formulation ensures that the decision boundary is determined by the support vectors, reflecting the critical points in the data that define the separation between classes.

2.5.2 Non-separable problems

In our prior discussions, we've proceeded on the premise that samples are linearly separable within the feature space. Yet, this isn't universally applicable, especially in scenarios with noisy data or other complexities. To address these challenges, we introduce the concept of error (represented by ξ_i) into our classification methodology.

With this definition, we can introduce the soft-margin optimization problem, which aims to minimize:

$$\frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^{N} \xi_n$$

subject to the constraints:

$$t_n \left(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_n) + b \right) \geq 1 - \xi_n \quad \forall n$$

where $\xi_n \geq 0$ are slack variables representing penalties for margin violations. The parameter C serves as a tradeoff between error and margin: it allows adjustment of the bias-variance tradeoff, and tuning may be necessary to find the optimal value for C.

Dual problem By obtaining the dual problem, we aim to maximize:

$$\tilde{\mathcal{L}}(\boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m)$$

subject to the constraints:

$$\begin{cases} 0 \leq \alpha_n \leq C \\ \sum_{n=1}^{N} \alpha_n t_n = 0 \end{cases} \qquad n = 1, \ldots, N$$

As usual, support vectors are the samples for which $\alpha_n > 0$. If $\alpha_n < C$, then $\xi_n = 0$, indicating that the sample is on the margin. If $\alpha_n = C$, the sample can be within the margin and either correctly classified ($\xi_n \leq 1$) or misclassified ($\xi_n > 1$).

Alternative formulation The same problem can be also formulated as the maximization of:

$$\tilde{\mathcal{L}}(\boldsymbol{\alpha}) = -\frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n \alpha_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m)$$

$$\begin{cases} 0 \leq \alpha_n \leq \frac{1}{N} \\ \sum_{n=1}^{N} \alpha_n t_n = 0 \\ \sum_{n=1}^{N} \alpha_n \geq \nu \qquad n = 1, \dots, N \end{cases}$$

Where $0 \leq \nu < 1$ is a user-defined parameter that enables control over both the margin errors and the number of support vectors, ensuring that the fraction of margin errors is less than or equal to ν and the fraction of support vectors is also less than or equal to ν.

2.5.3 Support vector machines training

To solve the optimization problem and find α_i and b, several methods exist. However, the direct solution is computationally expensive, typically $O(n^3)$ where n is the size of the training set.

To mitigate this computational burden, faster approaches have been developed, including:

1. Chunking: Breaking the problem into smaller chunks to solve separately.
2. Osuna's methods: Variants of chunking methods specifically tailored for SVM optimization.
3. Sequential Minimal Optimization (SMO): A method that optimizes the dual problem iteratively by selecting pairs of variables to update.

Additionally, for scenarios where online learning is preferred, methods such as chunking-based approaches and incremental methods can be employed. These methods update the model gradually as new data becomes available, thus avoiding the need to retrain the entire model from scratch.

Chunking Chunking solves iteratively by addressing a sub-problem known as the working set. The working set is constructed using the current support vectors and the M samples with the largest errors (known as the worst set). It's important to note that the size of the working set may dynamically increase during the iterations. Despite this, chunking converges to the optimal solution.

Osuna's Method Osuna's method also solves iteratively by focusing on a sub-problem, the working set. However, unlike chunking, it maintains a fixed size for the working set. This method replaces some samples in the working set with misclassified samples from the dataset. Despite its fixed-size working set, Osuna's method still converges to the optimal solution.

Sequential Minimal Optimization SMO operates iteratively, but uniquely, it only works on two samples at a time. By doing so, it keeps the size of the working set minimal. Moreover, the multipliers are found analytically during each iteration. Like the other methods, SMO converges to the optimal solution.

2.5.4 Multi-class Support vector machines

One against all In the one against all approach, a k-class problem is decomposed into k binary (2-class) problems. Training involves k SVM classifiers on the entire dataset. During testing, the class selected with the highest margin among the k SVM classifiers is chosen.

One against one In one against one, a k-class problem is decomposed into $\frac{k(k-1)}{2}$ binary problems. Here, $\frac{k(k-1)}{2}$ SVM classifiers are trained on subsets of the dataset. During testing, all $\frac{k(k-1)}{2}$ classifiers are applied to the new sample, and the most voted label is chosen.

DAGSVM DAGSVM also decomposes the k-class problem into $\frac{k(k-1)}{2}$ binary problems like one-against-one. However, it employs a Direct Acyclic Graph during testing to reduce the number of SVM classifiers to apply. This leads to only k-1 binary SVM classifiers being involved in the test process instead of $\frac{k(k-1)}{2}$ as in one-against-one.

Summary The methods are:

- One-against-all: requires less memory but has expensive training and cheap testing.
- One-against-one: requires more memory but has slightly cheaper training and expensive testing.
- DAGSVM: moderately expensive in terms of memory requirements, with slightly cheaper training and testing.

One-against-one is considered the best performing approach due to its effective decomposition. DAGSVM provides a faster approximation of one-against-one.

2.6 Computational learning theory

Computational learning theory is a field of study that aims to understand the general principles of inductive learning. It models the complexity of the hypothesis space, the bound on the training samples, the bound on accuracy, and the probability of successful learning.

A learner (L) aims to grasp a concept (C) that effectively relates data in the input space (X) to a target (t). Let's suppose L has identified a hypothesis h^* that perfectly fits the training data. We need to find how many training samples from X are required to ensure that L has genuinely acquired the true concept, meaning h^* accurately represents C.

Let $Acc(L)$ represent the generalization accuracy of learner L, indicating L's performance 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 learner L and any possible training set:

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

Corollary 2.6.0.1. *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 $Acc_G(L_2) > Acc_G(L_1)$.*

This means that in Machine Learning we always operate under some assumptions.

2.6.1 Approximately correct hypothesis

Let X be the instance space. Let $H = \{h : X \to \{0,1\}\}$ represent the hypothesis space of learner L. Let $C = \{c : X \to \{0,1\}\}$ denote the set of all possible target functions (concepts) we aim to learn. Let be $\mathcal{D}$ be the training data drawn from a stationary distribution $P(X)$ and labeled (without noise) according to a concept c we intend to learn. A learner L produces a hypothesis $h \in H$ such that:

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

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

$$error_{\mathcal{D}}(h) = \Pr_{x \in \mathcal{D}}[h(x) \neq c(x)] = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} I(h(x) \neq c(x))$$

This represents the training error. However, our interest lies in the true error of h:

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

Assuming $error_{true}$ as the probability of making a mistake on a sample, we can compute $error_{\mathcal{D}}$, which is the average error probability on $\mathcal{D}$. Assuming a Bernoulli distribution for the error probability, the 95% confidence interval is given by:

$$error_{true}(h) = error_{\mathcal{D}}(h) \pm 1.96\sqrt{\frac{error_{\mathcal{D}}(h)(1 - error_{\mathcal{D}}(h))}{n}}$$

This calculation is inaccurate because $\mathcal{D}$ represents the training data and is not independent of h. Therefore, we require a stricter bounding of the error under additional assumptions.

2.6.2 Version space and bound

A hypothesis 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 $\mathcal{D}$:

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

The version space, $VS_{H,\mathcal{D}}$, with respect to the hypothesis space H and the labeled dataset $\mathcal{D}$, is the subset of hypotheses in H consistent with $\mathcal{D}$:

$$VS_{H,\mathcal{D}} \overset{\text{def}}{=} \{h \in H | \text{Consistent}(h, \mathcal{D})\}$$

From now on, we consider only consistent learners, which always output a consistent hypothesis, i.e., a hypothesis in $VS_{H,\mathcal{D}}$, assuming it is not empty.

If we aim to bound the $error_{true}$ of a consistent learner, we need to find a bound for all the hypotheses in $VS_{H,\mathcal{D}}$.

Theorem 2.6.1. *If the hypothesis space 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 $VS_{H,\mathcal{D}}$ contains a hypothesis error greater than ε is less than $|H|\, e^{\varepsilon N}$:*

$$\Pr(\exists h \in H : error_{\mathcal{D}}(h) = 0 \land error_{true}(h) \geq \varepsilon) \leq |H|\, e^{\varepsilon N}$$

Proof. We have that:

$$\begin{aligned}
&\Pr\Big((error_{\mathcal{D}}(h_1) = 0 \land error_{true}(h_1) \geq \varepsilon) \lor \ldots \lor \Big(error_{\mathcal{D}}(h_{|VS_{H,\mathcal{D}}|}) = 0 \land error_{true}(h_{|VS_{H,\mathcal{D}}|}) \geq \varepsilon\Big)\Big) \\
&\quad \leq \sum_{h \in VS_{H,\mathcal{D}}} \Pr(error_{\mathcal{D}}(h) = 0 \land error_{true}(h) \geq \varepsilon) \\
&\quad \leq \sum_{h \in VS_{H,\mathcal{D}}} \Pr(error_{\mathcal{D}}(h) = 0 | error_{true}(h) \geq \varepsilon) \\
&\quad \leq \sum_{h \in VS_{H,\mathcal{D}}} (1-\varepsilon)^N \\
&\quad \leq |H|\,(1-\varepsilon)^N \\
&\quad \leq |H|\,e^{-\varepsilon N}
\end{aligned}$$

□

Bound in practice Let's denote δ as the probability of having $error_{true} > \varepsilon$ for a consistent hypothesis:

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

We can then bound N after setting ε and δ:

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

Similarly, we can bound ε after setting N and δ:

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

PAC-learning Considering a class C of possible target concepts defined over an instance space X with an encoding length M, and a learner L using an hypothesis space H we define: C is PAC-learnable by L using H if for all $c \in C$, for any distribution $\Pr(X)$, ε (such that $0 < \varepsilon < 1/2$), and δ (such that $0 < \delta < 1/2$), learner L will with a probability at least $(1-\delta)$ output a hypothesis $h \in H$ such that $error_{true}(h) \leq \varepsilon$, in time that is polynomial in $1/\varepsilon$, $1/\delta$, M, and $sixe(c)$. 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.

2.6.3 Agnostic learning

Up to this point, we've operated under the assumption that $c \in H$, or at the very least, that $VS_{H,\mathcal{D}}$ is not empty, and that the learner L will consistently output a hypothesis h such that $error_{\mathcal{D}}(h) = 0$. However, in a more general scenario, an agnostic learner might output a hypothesis h with $error_{\mathcal{D}}(h) > 0$.

Theorem 2.6.2. *If the hypothesis space 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 h, the probability that $error_{true}(h) - error_{\mathcal{D}}(h) > \varepsilon$ is less than $|H|\,e^{-2N\varepsilon^2}$:*

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

Proof. Utilizing the additive Hoeffding bound: let $\hat{\theta}$ be the empirical mean of N independent and identically distributed Bernoulli random variables with mean θ:

$$\Pr(\theta > \hat{\theta} + \varepsilon) \leq e^{-2N\varepsilon^2}$$

Consequently, for any single hypothesis h:

$$\Pr(error_{true}(h) > error_{\mathcal{D}}(h) + \varepsilon) \leq e^{-2N\varepsilon^2}$$

As we require this to hold true for all hypotheses in H:

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

□

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

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

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

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

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

Definition (*Dichotomy*)**.** A dichotomy of a set S of instances is defined as a partition of S into two disjoint subsets, i.e., labeling each instance in S as positive or negative.

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

The Vapnik-Chervonenkis dimension, $VC(H)$, of hypothesis space H over instance space X, is the largest finite subset of X shattered by H. If an arbitrarily large set of X can be shattered by H, then $VC(H) = \infty$.

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

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

CHAPTER 3

Model evaluation

3.1 Bias-variance framework

The bias-variance framework provides a structured approach for evaluating model performance.

Definition (*Data*)**.** Data are described as:

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

where $\mathbb{E}\left[\varepsilon\right] = 0$ and $\text{Var}\left[\varepsilon\right] = \sigma^2$.

Definition (*Model*)**.** The model is represented as:

$$\hat{t}_i = y(\mathbf{x}_i) + \varepsilon$$

learned from a sampled dataset $\mathcal{D} = \{\mathbf{x}_i, t_i\}$.

Definition (*Performance*)**.** Performance is quantified by:

$$\mathbb{E}\left[(t - y(\mathbf{x}))^2\right]$$

which measures the expected squared error.

Hence, the expected squared error can be decomposed as follows:

$$\begin{aligned}
\mathbb{E}\left[(t - y(\mathbf{x}))^2\right] &= \mathbb{E}\left[\left(t^2 + y(\mathbf{x})^2 - 2ty(\mathbf{x})\right)\right] \\
&= \mathbb{E}\left[t^2\right] + \mathbb{E}\left[y(\mathbf{x})^2\right] - \mathbb{E}\left[2ty(\mathbf{x})\right] \\
&= \mathbb{E}\left[t^2\right] + \mathbb{E}[t]^2 - \mathbb{E}[t]^2 + \mathbb{E}\left[y(\mathbf{x})^2\right] + \mathbb{E}[y(\mathbf{x})]^2 - \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}\left[y(\mathbf{x})\right] \\
&= \text{Var}\left[t\right] + \mathbb{E}[t]^2 + \text{Var}\left[y(\mathbf{x})\right] + \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}\left[y(\mathbf{x})\right] \\
&= \text{Var}\left[t\right] + \text{Var}\left[y(\mathbf{x})\right] + \left(f(\mathbf{x}) - \mathbb{E}\left[y(\mathbf{x})\right]\right)^2 \\
&= \underbrace{\text{Var}\left[t\right]}_{\sigma^2} + \underbrace{\text{Var}\left[y(\mathbf{x})\right]}_{\text{variance}} + \underbrace{\mathbb{E}[f(\mathbf{x}) - y(\mathbf{x})]^2}_{\text{squared bias}}
\end{aligned}$$

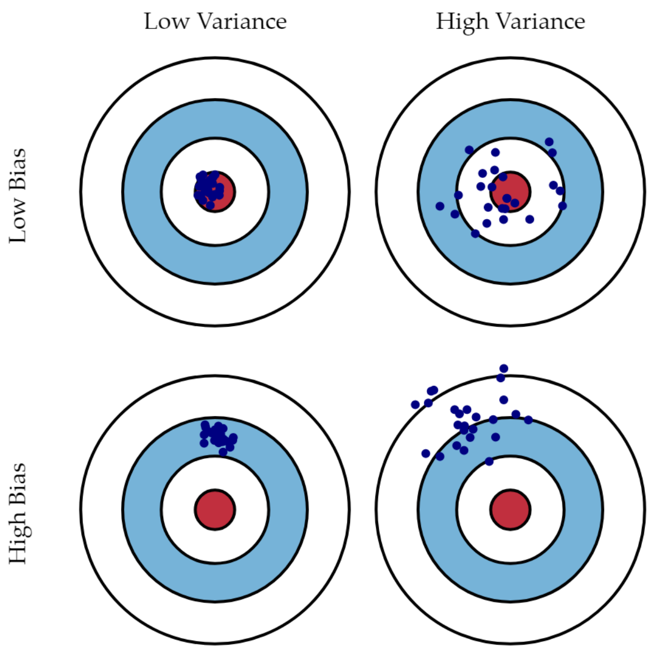


Figure 3.1: Bias-variance framework

Model variance When we sample multiple datasets $\mathcal{D}$, we obtain distinct models $y(\mathbf{x})$. Variance quantifies the dissimilarity between each model learned from a specific dataset and our anticipated learning outcome:

$$\text{variance} = \int \mathbb{E}\left[(y(\mathbf{x} - \bar{y}(\mathbf{x})))^2\right] \mathrm{P}(\mathbf{x})d\mathbf{x}$$

The variance diminishes by simplifying the model or increasing the sample size.

Model bias Bias gauges the disparity between the truth (f) and our expected learning outcome $(\mathbb{E}\left[y(\mathbf{x})\right])$:

$$\text{bias}^2 = \int (f(\mathbf{x}) - \bar{y}(\mathbf{x}))^2 \mathrm{P}(\mathbf{x})d\mathbf{x}$$

Bias decreases with more complex models.

Definition (*Data noise*)**.** Data noise (σ^2) represents the variance of data and remains constant regardless of data sampling or model complexity.

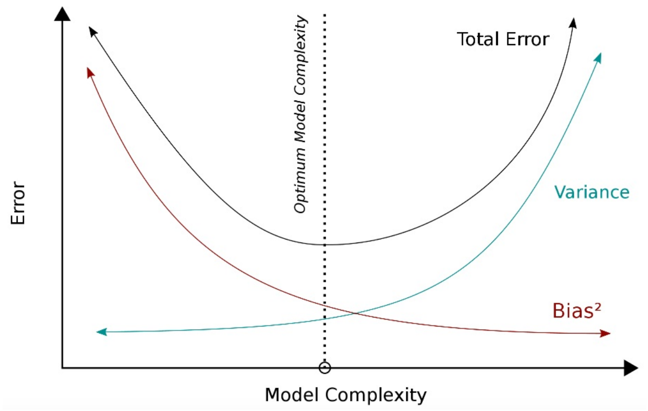
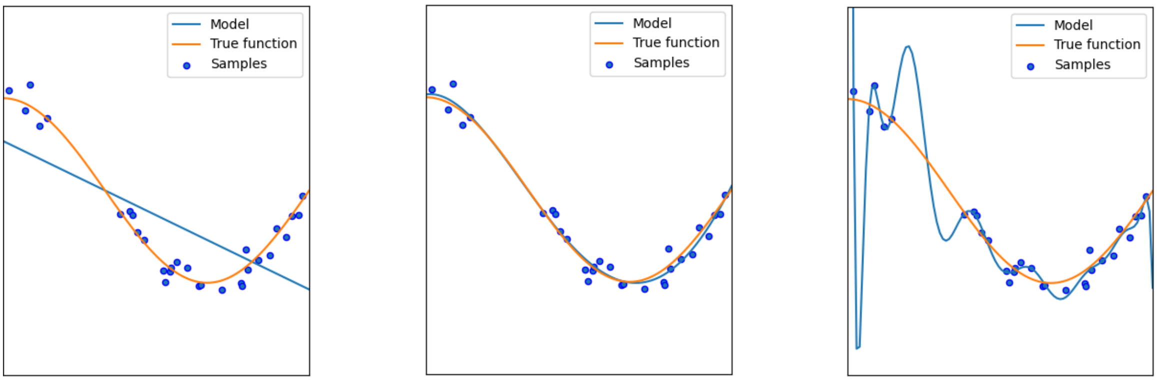


Figure 3.2: Bias-variance framework impact

In practical terms, the estimation is affected as follows:

- High variance leads to overfitting.
- High bias results in underfitting.
- Low bias and low variance yield a well-balanced approximation.



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

Figure 3.3: Bias-variance balancing

3.1.1 Regularization and bias-variance

The bias-variance decomposition elucidates why regularization enhances error reduction on unseen data. Lasso surpasses ridge regression when only a few features are linked to the output.

3.2 Model assessment

Given a dataset $\mathcal{D} = \{\mathbf{x}_i, t_i\}$ with $i = 1, \dots, N$, we can choose a model based on the computed loss L on $\mathcal{D}$. For regression, the loss function is defined as:

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

And for classification, the loss function becomes:

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

The training error decreases as the model complexity increases.

However, it's important to note that the training error doesn't give an accurate estimate of the error on new data, known as the prediction error. For regression, the prediction error is represented as:

$$L_{true} = \iint (t - y(\mathbf{x}))^2 \mathrm{P}(\mathbf{x}, t) d\mathbf{x} dt$$

And for classification, it is:

$$L_{true} = \iint I(t \neq y(\mathbf{x}))\mathrm{P}(\mathbf{x}, t) d\mathbf{x} dt$$

Unfortunately, modeling the joint probability distribution $\mathrm{P}(\mathbf{x}, t)$ is often not feasible.

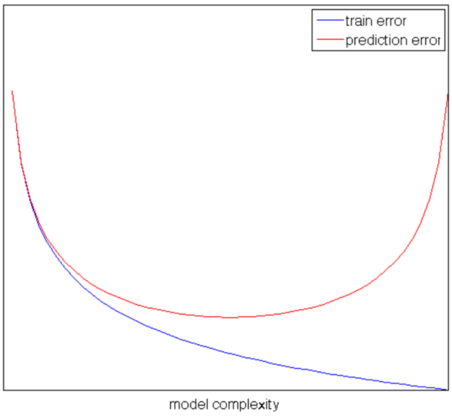


Figure 3.4: Train error compared to prediction error

Practical application In practical scenarios, data is typically randomly split into a training set and a test set. Model parameters are optimized using the training set, and the prediction error is estimated using the test set. For regression, this estimation yields:

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

And for classification:

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} I(t_n \neq y(\mathbf{x}_n))$$

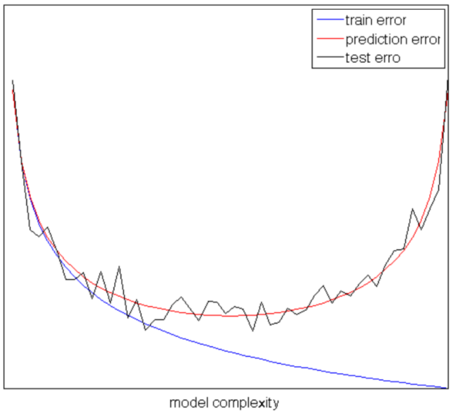


Figure 3.5: Error in practice

As the number of data points increases, these errors tend to converge, as depicted in the following figure:

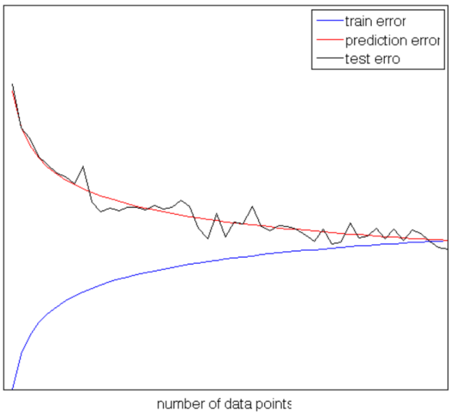


Figure 3.6: Error in function of the number of data points

Analyzing the train-test errors helps identify potential issues:

- *High bias*: when both training and test errors are higher than expected and close to each other.
- *High variance*: when the training error is significantly lower than expected and gradually approaches the test error.

Problems Frequently, data availability is constrained, and the test error tends to be minimal, leading to potential overestimation or underestimation of prediction error. Utilizing test error for model selection can result in overfitting to the test set. An unbiased estimate of prediction error is achievable only if the test set remains separate from both training and model selection phases.

3.2.1 Optimal model

To select the optimal model and determine the appropriate hyperparameters, we initially partition the data into three subsets: training data, validation data, and test data. The process involves the following steps:

1. Utilize the training data to train the model parameters.
2. For each trained model, assess its performance using the validation data to compute the validation error.
3. Identify the model with the lowest validation error, and subsequently, employ the test data to estimate the prediction error.

However, for this approach to be dependable, it's imperative that the validation data set is sufficiently sizable, especially in comparison to the training data set. Otherwise, there's a risk of overfitting to the validation data, potentially leading to the selection of a suboptimal model.

Leave-one-out cross validation Leave-one-out cross-validation (LOO-CV) involves training the model on all samples in the dataset $\mathcal{D}$ except for a single sample $\{\mathbf{x}_i, t_i\}$, and then evaluating the model's performance on that omitted sample. The prediction error estimate of our model is then computed as the average error across all single-sample evaluations:

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

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

The L_{LOO} estimate of prediction error provides an almost unbiased assessment (slightly pessimistic). However, LOO-CV is computationally intensive due to its requirement to repeatedly train models on nearly all data points.

K-fold cross validation K-fold cross-validation involves randomly dividing the training data $\mathcal{D}$ into k 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 then the error is computed on $\mathcal{D}_i$ as follows:

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

Finally, the prediction error is estimated as the average error computed across all folds:

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

The $L_{k\text{-fold}}$ estimate of prediction error provides a slightly biased (pessimistic) assessment but is computationally less expensive. Typically, k is set to ten.

Other metrics Various metrics are employed to evaluate models by adjusting their training error based on their complexity:

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

Here, M represents the number of parameters, N denotes the number of samples, L signifies the loss function, σ^2 stands for the estimate of noise variance, RSS corresponds to the residual sum of squares, and TSS indicates the total sum of squares.

AIC and BIC are typically utilized when maximizing the log-likelihood. BIC tends to penalize model complexity more severely compared to AIC.

3.3 Model complexity

Introducing an additional feature leads to an exponential growth in the volume of the input space. This growth leads to the following problems:

- *Computational cost*: the computational resources required to process and analyze the expanded input space increase significantly.
- *Data quantity*: the amount of data needed to effectively explore and train models in the expanded input space may be substantial.
- *Large model variance* (overfitting): with the increased complexity of the input space, there is a higher risk of models capturing noise or irrelevant patterns, leading to overfitting and decreased generalization performance.

Our goal is to choose the model with the minimal prediction error, which can be attained by decreasing the variance of the model:

- *Feature selection*: by carefully designing the feature space, we can choose the most impactful subset from all available features.
- *Dimensionality reduction*: mapping the input space to a lower-dimensional representation can effectively reduce complexity and variance.
- *Regularization*: shrinkage of parameter values towards zero helps control model complexity and mitigate overfitting.

These approaches are not mutually exclusive and can be combined to enhance model performance.

3.3.1 Feature selection

The most straightforward approach appears to be comparing all possible combinations of features. Given M features, for each $k = 1, \dots, M$, we would need to train all $\binom{M}{k} = \frac{M!}{k!(M-k)!}$ models with exactly k features and select the optimal one. However, this procedure quickly becomes computationally impractical.

In practical scenarios, feature selection is often carried out based on the specific model being utilized:

- *Filter*: features are assessed individually using certain evaluation metrics (e.g., correlation, variance, information gain), and the top k features are selected. While this method is very fast, it fails to capture any subset of mutually dependent features.
- *Embedded*: feature selection is integrated into the Machine Learning approach itself (e.g., lasso, decision trees). Although this method is not computationally expensive, the features identified are specific to the chosen learning approach.
- *Wrapper*: a search algorithm is employed to identify a subset of features by iteratively training a model with different feature subsets and evaluating their performance. This method utilizes either a simpler model or a basic Machine Learning approach to evaluate the features. Greedy algorithms are typically employed to search for the best feature subset.

3.3.2 Dimensionality reduction

Dimensionality reduction aims to decrease the dimensions of the input space, but it differs from feature selection in two significant ways:

- It utilizes all features and transforms them into a lower-dimensional space.
- It is an unsupervised approach, meaning it doesn't rely on labeled data for training.

There are numerous methods for performing dimensionality reduction, including:

- Principal Component Analysis (PCA).
- Independent Component Analysis (ICA).
- Self-Organizing Maps.
- Autoencoders.
- ISOMAP.
- t-SNE.

3.4 Ensemble

We've explored methods to decrease variance while balancing increased bias. However, we want to reduce variance without amplifying bias or mitigate bias altogether.

These objectives can indeed be achieved through the utilization of two ensemble methods involving the learning of multiple models and their combination:

- *Bagging*: involves training multiple models independently on different subsets of the data and then combining their predictions.
- *Boosting*: utilizes an iterative approach where models are sequentially trained, each aiming to correct the errors of its predecessors, leading to the creation of a strong ensemble model.

3.4.1 Bagging

Let assume to have N datasets and to learn from them N models, $y_1, y_2, \ldots, y_N$. Now let us compute an aggregate model as:

$$y_{AGG} = \frac{1}{N} \sum_{i=1}^{N} y_i$$

If the datasets are independent, the model variance of y_{AGG} will be $\frac{1}{N}$ of the model variance of the single model y_i. However, we generally do not have N datasets.

Bagging, short for Bootstrap Aggregation, involves the following steps:

1. Generate N datasets by applying random sampling with replacement.
2. Train a model (classification or regression) using each dataset generated.

3. To predict new samples, apply all the trained models and combine their outputs using majority voting (for classification) or averaging (for regression).

Bagging is generally beneficial as it reduces variance, although the sampled datasets are not independent. It proves particularly useful with unstable learners, characterized by significant changes with small dataset variations (low bias and high variance), and in scenarios with a high degree of overfitting (low bias and high variance). However, it does not offer much help with robust learners, which are insensitive to data changes (typically higher bias but lower variance).

3.4.2 Boosting

Boosting aims to minimize bias by employing a series of simple (weak) learners.

The core concept of boosting involves iteratively training a sequence of weak learners, with each iteration concentrating on the samples misclassified in the preceding iteration.

Ultimately, an ensemble model is constructed by combining the outputs of all the weak learners trained.

3.4.3 Summary

The characteristics of bagging are:

- Decreases variance.
- Less effective for stable learners.
- Applicable with noisy data.
- Generally beneficial, though the improvement might be modest.
- Naturally suited for parallelization.

The characteristics of boosting are:

- Reduces bias (typically without overfitting).
- Compatible with stable learners.
- May encounter challenges with noisy data.
- Not always effective, but can yield significant improvements.
- Sequential in nature.

CHAPTER 4

Reinforcement learning

4.1 Introduction

In reinforcement learning we train a model by providing it with an evaluation of its output

Sequential decision making In the realm of sequential decision making, we navigate through a series of choices or actions aimed at achieving a specific objective. The optimal actions are contingent upon the context in which they occur, often lacking clear-cut examples of correctness. Moreover, these actions can yield long-term ramifications, and while the short-term outcomes of optimal decisions may appear unfavorable, they serve a greater purpose in the pursuit of our goals.

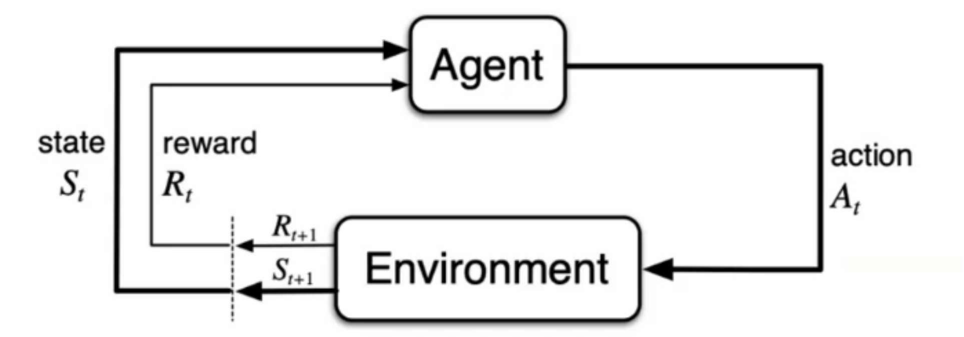


Figure 4.1: Agent-environment interface

At discrete time steps $t = 0, 1, 2, K$, the agent and environment interact as follows: the agent observes the state at step t, denoted as $S_t \in \mathcal{S}$, produces an action at step t, represented by $A_t \in \mathcal{A}(S_t)$, gets the resulting reward $R_{t+1} \in \mathcal{R}$, and transitions the environment to the next state $S_{t+1} \in \mathcal{S}$.

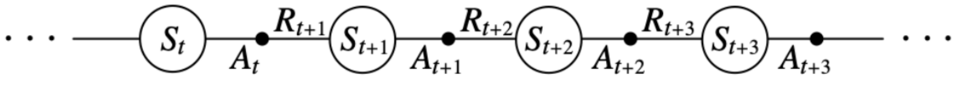


Figure 4.2: Agent-environment interface

4.2 Markov decision process

Markov decision processes adhere to Markov property:

Property 4.2.1. The future state (s') and reward (r) only depend on current state (s) and action (a)

It is not a limiting assumption, it can be seen as a property of state

One-step dynamic In a Markov Decision Process (MDP), the one-step dynamic can be described as:

$$p(s', r|s, a)$$

That is defined on: $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$. The main property is that:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1 \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$$

4.2.1 Finite Markov decision processes

When the Markov Property holds and both the state and action sets are finite, the problem is termed as a finite Markov Decision Process. To formally define a finite MDP, it's necessary to specify the following: sets for states and actions, and one-step dynamics:

$$p(s', r|s, a) = \Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\}$$

We can further deduce the distribution of the next state and the expected reward: The overarching formulation is as follows:

$$p(s'|s, a) \doteq \Pr\{S_{t+1} = s'|S_t = s, A_t = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a)$$
$$r(s, a) \doteq \mathbb{E}\left[R_{t+1}|S_t = s, A_t = a\right] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a)$$

Return The agent should refrain from selecting actions solely based on immediate rewards. Instead, prioritizing long-term consequences over short-term gains is crucial. Hence, it's imperative to consider the sequence of future rewards. To this end, we define the return, G_t, as a function of the sequence of future rewards.

$$G_t \doteq f\left(R_{t+1} + R_{t+2} + R_{t+3} + \cdots\right)$$

To achieve success, the agent must aim to maximize the expected return, denoted as $\mathbb{E}\left[G_t\right]$. Various definitions of return are conceivable, including: total reward, discounted reward, or average reward.

Episodic task In episodic task the agent-environment interaction naturally breaks into chunks called episodes

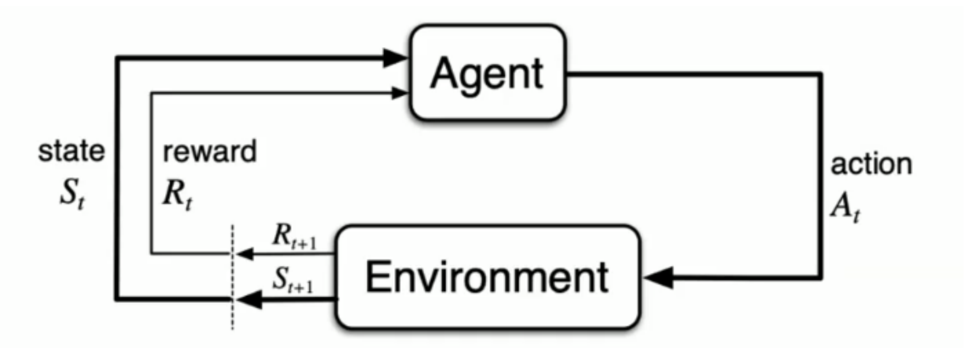


Figure 4.3: Markov decision processes episodic tasks

It is possible to maximize the expected total reward:

$$\mathbb{E}\left[G_t\right] = \mathbb{E}\left[R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T\right]$$

Continuing tasks In continuing task the agent-environment interaction goes on continually and there are no terminal state The total reward is a sum over an infinite sequence and might not be finite:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_{t+k} + \cdots \overset{?}{=} \infty$$

To solve this issue we can discount the future rewards by a factor γ $(0 < \gamma < 1)$:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{k-1} R_{t+k} + \cdots \overset{<}{\infty}$$

Thus, the expected reward to maximize will be defined as:

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

Return general notation In episodic tasks, we number from zero the time steps for each episode We can design terminal state as absorbing states that always produce zer. reward: We can use the same definition of expected reward for episodic and continuing tasks:

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

Where:

- $\gamma = 1$ can be used if an absorbing state is always reached.
- $\gamma = 0$, agent would only care about immediate reward.
- As $\gamma \to 1$, agent would take future rewards into account more strongly.

Goal and reward A goal should state what we want to achieve, not how we want to achieve it. This idea aligns with the Reward Hypothesis, suggesting that our goals can be seen as maximizing the expected cumulative sum of received rewards.

4.2.2 Policy

A policy, at any particular moment, determines the action that the agent selects. It entirely characterizes the behavior of an agent. Policies can vary in several dimensions:

- Markovian or non-Markovian.
- Deterministic or stochastic.
- Stationary or non-Stationary.

Deterministic policy In its simplest form, the policy can be represented as a function ($\pi : \mathcal{S} \rightarrow \mathcal{A}$):

$$\pi(s) = a$$

In this setup, the policy directly maps each state to a specific action. Such policies can be effectively depicted using a table.

Stochastic policy A more versatile approach involves modeling the policy as a function that assigns each state a probability distribution over the available actions:

$$\pi(a|s)$$

Where:

- $\sum_{a \in \mathcal{A}(s)} \pi(a|s) = 1$.
- $\pi(a|s) \geq 0$

A stochastic policy can accommodate deterministic policies as well.

4.2.3 Value functions

For a given policy π, we can compute the state-value function as:

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

This function signifies the expected return from a specific state s, following policy π,

Similarly, we can calculate the action-value function as:

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

This function represents the expected return from a given state s when a particular action a is chosen, followed by policy π.

Bellman expectation equation The state-value function can again be decomposed into immediate reward plus discounted value of successor state:

$$V_\pi(s) \doteq \mathbb{E}\left[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s\right] = \sum_{a \in \mathcal{A}} \pi(a|s) \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s') \right]$$

The action-value function can be similarly decomposed:

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi\left[R_{t+1} + \gamma V_\pi\left(S_{t+1}\right) | S_t = s, A_t = a\right] \\ &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s') \\ &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s',a') \end{aligned}$$

4.2.4 Optimality

We denote that $\pi \geq \pi'$ if and only if $V_\pi(s) \geq V'_\pi(s)$ for all $s \in \mathcal{S}$. For any Markov Decision Process, there always exists at least one optimal deterministic policy π^* that is superior or equal to all others:

$$\pi^* \geq \pi \qquad \forall \pi$$

This occurs because we can select different policies for each interval, consistently choosing the optimal one. In Markov decision processes, we have $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies, making brute force search computationally infeasible.

Optimal Value Function To solve this computational problem we can use the optimal value function. Given the optimality definition for the policy, we can compute optimal state-value function and optimal action-value function as:

$$\begin{aligned} V^*(s) &\doteq \max_\pi V_\pi(s) \qquad \forall s \in \mathcal{S} \\ Q^*(s,a) &\doteq \max_\pi Q_\pi(s,a) \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \end{aligned}$$

The corresponding Bellman Optimality Equation for $V^*(s)$ is:

$$\begin{aligned} V^*(s) &= \sum_{a \in \mathcal{A}} \pi^*(a|s) \left(r(s,a) + \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s') \right) \\ &= \max_a \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s') \right\} \end{aligned}$$

The corresponding Bellman Optimality Equation for $Q^*(s)$ is:

$$\begin{aligned} Q^*(s,a) &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \sum_{a' \in \mathcal{A}} \pi^*(a'|s') Q^*(s',a') \\ &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \max_{a'} Q^*(s',a') \end{aligned}$$

We can make this change since the considered policy is optimal. From $V^*(s)$ and $Q^*(s,a)$ we can easily compute the optimal policy π^* as:

$$V^*(s) = \underset{a}{\operatorname{argmax}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s') \right\}$$

The problem with this function is that it is not linear. To make it linear we again need the value of π^*, and since it is computationally infeasible to compute this function for every policy we need a different approach.

4.3 Dynamic programming

To resolve an MDP, locating the optimal policy is essential. However, employing a brute force method is impractical due to the necessity to solve $|\mathcal{S}|$ linear equations for each policy. Dynamic Programming (DP) offers a solution by dissecting the intricate problem into more manageable sub-problems recursively. Through the utilization of DP, we'll explore how to effectively tackle an MDP using the Bellman Equations.

By utilizing Dynamic programming we can evaluate multiple policies and compute the corresponding state-value function. Then, by using the Bellman equation we can find the optimal one again using dynamic programming.

4.3.1 Policy evaluation

We search the solution of the Bellman expectation equation:

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

DP solves this problem through iterative application of Bellman equation:

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

At each iteration k, the value-function V_k is updated for all state $s \in \mathcal{S}$. It can be proved that V_k converge to V_π as k tends to infinity for any initial value V_0.

Algorithm 2 Iterative policy evaluation algorithm

```
Initialize $V(s)$ for all $s \in \mathcal{S}^+$ arbitrarily
$V(\text{terminal}) = 0$
repeat
    $\Delta = 0$
    for each $s \in \mathcal{S}$ do
        $v = V(s)$
        $V(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$
        $\Delta = \max(\Delta, |v - V(s)|)$
    end for
until $\Delta < \theta$
```

The input of this algorithm is the policy to be evaluated π. It also has a small threshold $\theta > 0$, that is a parameter used to determine the accuracy of the estimation.

4.3.2 Policy improvement

Normally, the optimal policy from optimal value functions is derived as:

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

If we act greedy with respect to non optimal value function we have:

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

We may have two outcomes:

- $\pi' = \pi$ it means that π is already the optimal policy π^*.
- $\pi' \neq \pi$ it means that π' is better or as good as π.

The second point is guaranteed by the following theorem.

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

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

Then π' is better or as good as π:

$$\pi' \geq \pi$$

Corollary 4.3.1.1. *If exists $s \in \mathcal{S}$ such that $Q_\pi(s, \pi'(s)) > Q_\pi(s, \pi(s))$, then $\pi' > \pi$.*

Proof. We have that:

$$\begin{aligned} V_\pi(s) &\leq Q_\pi(s, \pi(s')) = \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma Q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 Q_\pi(S_{t+2}, \pi'(S_{t+2})) | S_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \cdots | S_t = s\right] = V_{\pi'}(s) \end{aligned}$$

□

4.3.3 Policy iteration

We can exploit the policy improvement theorem to find the optimal policy:

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^*$$

Algorithm 3 Policy iteration algorithm

```
$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$          ▷ Initialization
repeat
    repeat                                                                   ▷ Policy evaluation
        $\Delta = 0$
        for each $s \in \mathcal{S}$ do
            $v = V(s)$
            $V(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V(s')\right]$
            $\Delta = \max\left(\Delta, |v - V(s)|\right)$
        end for
    until $\Delta < \theta$
    policy-stable = true                                                     ▷ Policy improvement
    for each $s \in \mathcal{S}$ do
        old-action = $\pi(s)$
        $\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        if old-action $\neq \pi(s)$ then
            policy-stable = false
        end if
    end for
until policy-stable = true
return $V \approx v^*$ and $\pi \approx \pi^*$
```

4.3.4 Value iteration

Policy iteration alternates complete policy evaluation and improvement up to the convergence Policy iteration framework allows also to find the optimal policy interleaving partial evaluation and improvement steps In particular, Value Iteration is one of the most popular GPI method In the policy evaluation step, only a single sweep of updates is performed:

$$\pi'(s) = \underset{a}{\text{argmax}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s') \right\} \qquad \forall s \in \mathcal{S}$$

$$V_{k+1}(s) = \sum_a \in \mathcal{A}\pi'(a|s) \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s') \right) \qquad \forall s \in \mathcal{S}$$

Combining them, we simply need to iterate the update of the value function using the Bellman optimality equation:

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

It can be proved that:

$$\lim_{k \to \infty} V_k = V^*$$

Algorithm 4 Iterative policy evaluation algorithm

```
Initialize $V(s)$ for all $s \in \mathcal{S}^+$ arbitrarily
$V(\text{terminal}) = 0$
repeat
    $\Delta = 0$
    for each $s \in \mathcal{S}$ do
        $v = V(s)$
        $V(s) = \max_a \sum_{s',r} p(s', r|s, a) [r + \gamma V(s')]$
        $\Delta = \max(\Delta, |v - V(s)|)$
    end for
until $\Delta < \theta$
```

It also has a small threshold $\theta > 0$, that is a parameter used to determine the accuracy of the estimation. The output is a deterministic policy $\pi \approx \pi^*$, such that:

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

4.3.5 Efficiency

All previously described DP methods mandate exhaustive sweeps across the complete state set. However, Asynchronous DP diverges from this approach by eschewing sweeps. Instead, it operates by selecting a state randomly, applying the relevant backup, and iterating until a convergence criterion is satisfied. We can chose states for backup in a more intelligent manner by noticing that an agent's experience can serve as a valuable guide in this regard.

The complexity of finding an optimal policy is polynomial in the number of states and actions:

- For value iteration: $O(|\mathcal{S}|^2 |\mathcal{A}|)$.
- For policy iteration:
  - Iterative evaluation: $O\left(\frac{|\mathcal{S}|^2 \log\left(\frac{1}{\epsilon}\right)}{\log\left(\frac{1}{\gamma}\right)}\right)$
  - Improvement: $O\left(\frac{|\mathcal{A}|}{1-\gamma} \log\left(\frac{|\mathcal{S}|}{1-\gamma}\right)\right)$

Unfortunately, the number of states can become extremely large, often growing exponentially with the number of state variables (known as the curse of dimensionality). Classical DP works well for problems with a few million states, but Asynchronous DP can handle larger ones and is suitable for parallel computing. However, there are MDPs where DP methods become impractical. Linear programming approaches are an alternative, but they don't scale well for larger problems.

4.4 Monte Carlo methods

Dynamic Programming enables us to determine the optimal value function and corresponding optimal policy. However, its major limitation lies in the assumption that we have full knowledge of the problem dynamics. To overcome this limitation, we seek methods that can learn the optimal policy directly from data.

Monte Carlo methods rely solely on experience (data) to learn value functions and policies. They can be utilized in two ways:

- *Model-free*: no model is necessary, yet it can still achieve optimality.
- *Simulated*: requires only a simulation, not a complete model.

Monte Carlo methods learn from complete sample returns and are exclusively defined for episodic tasks.

4.4.1 Policy evaluation

The goal of Monte Carlo policy evaluation is to learn $V_\pi(s)$ given some number of episodes under π which contain s. The idea is to average the returns observed after visits to s:

$$V_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] \rightarrow V_\pi(s) \approx \text{average}[G_t|S_t = s]$$

We can perform Monte Carlo policy evaluation in two ways:

- Every-Visit MC: average returns for every time s is visited in an episode
- First-visit MC: average returns only for first time s is visited in an episode

Note that Both converge asymptotically

Algorithm 5 Monte Carlo policy evaluation algorithm

```
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 $\pi : S_0, A_0, R_1S_1, A_1, R_2, \cdots, S_{T-1}, A_{T-1}, R_T$
6:         $G = 0$
7:         for each step of episode $t = T-1, T-2, \cdots, 0$ do
8:             $G = \gamma G + R_{t+1}$
9:             if $S_t \notin \{S_0, S_1. \cdots, S_{t-1}\}$ 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
```

The input of this algorithm is a policy π to be evaluated. The incremental updates of lines ten and eleven can be done in the following way:

$$N(S_t) = N(S_t) + 1$$

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)}(G - V(S_t)) \text{ or } V(S_t) = V(S_t) + \alpha(G_t - V(S_t))$$

4.4.2 Policy iteration

To improve the policy we need to find a policy that maximized the q value function:

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

To do so, we average return starting from state s and action a following π:

$$Q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \to Q_\pi(s,a) \approx \text{average}[G_t|S_t = s, A_t = a]$$

This method Converges asymptotically if every state-action pair is visited.

To have this full exploration in a simple way we use exploring starts. We choose randomly the first state and the first action, and we perform the following algorithm.

Algorithm 6 Monte Carlo exploring starts

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

4.4.3 Epsilon-soft Monte Carlo policy iteration

Exploring starts is a simple idea but it is not always possible. But, we need to keep exploring during the learning process This leads to a key problem in RL: the Exploration-Exploitation Dilemma

ε-Greedy Exploration is the simplest solution to the exploration-exploitation dilemma Instead of searching the optimal deterministic policy we search the optimal ε-soft policy, i.e., a policy that selects each action with a probability that is at least $\frac{\varepsilon}{|\mathcal{A}|}$.

In particular we use ε-greedy policy:

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

This algorithm takes as input a small $\varepsilon > 0$

Algorithm 7 ε-soft Monte Carlo policy iteration

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

Proof. We have that:

$$\begin{aligned} V_\pi(s) &= Q_\pi(s, \pi'(s)) \\ &= \sum_{a \in \mathcal{A}} \pi'(a|s) Q_\pi(s,a) \\ &= \varepsilon \sum_{a \in \mathcal{A}} \frac{1}{|A|} \sum_{a \in \mathcal{A}} Q_\pi(s,a) + (1-\varepsilon) \max_{a \in \mathcal{A}} Q^\pi(s,a) \\ &\geq \varepsilon \sum_{a \in \mathcal{A}} Q_\pi(s,a) + (1-\varepsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a|s) - \frac{\varepsilon}{|A|}}{1-\varepsilon} \bar{Q}_\pi(s,a) \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s,a) = V_\pi(s) \end{aligned}$$

□

4.4.4 Off-policy learning

On-policy learning On-policy learning involves the agent learning the value functions based on the same policy it uses to select actions. This method faces challenges in balancing exploration and exploitation, making it difficult to converge to an optimal deterministic policy.

Off-policy learning Off-policy learning, on the other hand, allows the agent to select actions using a behavior policy $b(a|s)$, while learning the value functions of a different target policy

$\pi(a|s)$. This flexibility enables the agent to use an explorative behavior policy, while still learning towards an optimal deterministic policy $\pi^*(a|s)$.

Regardless of our behavior policy, it's impossible to learn any policy $\pi(a|s)$ if there are actions in that state with zero probability according to the behavior policy $b(a|s)$. This situation occurs when the behavior policy never transitions to a particular state from the current one.

Importance sampling Importance sampling enables the estimation of expectations of a distribution that differs from the one used to draw the samples:

$$\mathbb{E}_p[x] = \sum_{x \in X} xp(x) = \sum_{x \in X} x\frac{p(x)}{q(x)}q(x) = \sum_{x \in X} z\rho(x)q(x) = \mathbb{E}_q[x\rho(x)]$$

Consequently, for sample-based estimation:

$$\mathbb{E}_p[x] \approx \frac{1}{N}\sum_{i=1}^{N} x_i \text{ if } x_i \sim p(x) \rightarrow \mathbb{E}_p[x] \approx \frac{1}{N}\sum_{i=1}^{N} x_i\rho(x_i) \text{ if } x_i \sim q(x)$$

Importance sampling in policy evaluation When adhering to policy π, the computation of the state value function is expressed as:

$$V_\pi(s) \approx \text{average}(\text{Returns}[0], \text{Returns}[1], \text{Returns}[2], \cdots)$$

However, under policy b, the value function transforms into:

$$V_\pi(s) \approx \text{average}(\rho_0\text{Returns}[0], \rho_1\text{Returns}[1], \rho_2\text{Returns}[2], \cdots)$$

Here, ρ_i denotes the probability of executing the trajectory observed in episode i while adhering to policy π, relative to the probability of observing the same trajectory while following policy b:

$$\rho = \frac{\text{Pr(trajectory under } \pi)}{\text{Pr(trajectory under } b)}$$

In practical terms, $\rho_{t:T-1}$ can be calculated as:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

Sampling methods include:

- *Ordinary*: unbiased with higher variance:

$$V_\pi(s) \approx \frac{\sum_i \rho[i]\text{Return}[i]}{N(s)}$$

- *Weighted*: biased (bias converges to zero) with lower variance:

$$V_\pi(s) \approx \frac{\sum_i \rho[i]\text{Return}[i]}{\sum_i \rho[i]}$$

Algorithm 8 Off-Policy every visit Monte Carlo prediction

```
$V(s) \in \mathbb{R}$ arbitrarily, for all $s \in S$
Returns($s$) = an empty list, for all $s \in S$
**for** each episode **do**
    Generate an episode following $b : S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G = 0$
    $W = 1$
    **for** each step of episode, $t = T-1, T-2, \ldots, 0$ **do**
        $G = \gamma W G + R_{t+1}$
        Append $G$ to Returns($S_t$)
        $V(S_t)$ = average(Returns($S_t$))
        $W = W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
    **end for**
**end for**
```

The input of this algorithm is a policy π to be evaluated.

4.5 Multi-armed bandits

In the k-armed bandit problem, an agent faces a decision-making scenario where it selects from k actions and receives a reward based on the chosen action. The objective is to identify the optimal action among the available options. Unlike many decision problems, this setting lacks contextual information; decisions are made in isolation, without considering a broader state.

Feedback in this problem manifests as evaluations (rewards) of decisions made under uncertainty, with learning occurring through trial and error and interaction with the environment.

The value associated with each action is represented by its expected reward:

$$q^* \doteq \mathbb{E}[R_t|A_t = a] = \sum p(r|a)r \qquad \forall a \in \{1, \cdots, k\}$$

Here, the agent's aim is to maximize the expected reward by selecting:

$$\operatorname*{argmax}_a q^*(a)$$

Since the exact distribution $p(r|a)$ is typically unknown, the agent estimates $q^*(a)$ based on its experiences:

$$Q_t(a) \doteq \frac{\sum_{i=1}^{t-1} R_i \nVdash_{A_t=a}}{\sum_{i=1}^{t-1} \nVdash_{A_t=a}}$$

This expression represents the ratio of the cumulative rewards received when action a was chosen before time step t, divided by the number of times action a was selected up to time step t.

4.5.1 Incremental update of action-values

Let's examine the update for a single action:

$$\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)Q_n\right) \\
&= Q_n + \frac{1}{n}\left(R_n - Q_n\right)
\end{aligned}$$

In this equation, Q_{n+1} represents the new estimate, Q_n denotes the old estimate, $\frac{1}{n}$ stands for the step size, and $(R_n - Q_n)$ serves as the target for the old estimate.

Non-stationary bandit problem For non-stationary bandit problems, the update equation takes the form:

$$Q_{n+1} = Q_n + \alpha\left(R_n - Q_n\right)$$

Here, the parameter α varies over time.

4.5.2 Epsilon-greedy action selection

Selecting the action with the highest value isn't always optimal, as it may not lead to the best outcome. Thus, striking a balance between exploration and exploitation becomes crucial:

- Exploitation: The agent leverages its current knowledge to gain immediate rewards.
- Exploration: The agent seeks to enhance its knowledge for long-term gains.

To navigate this trade-off, we can employ epsilon-greedy action selection:

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

4.5.3 Optimistic initial values

Traditionally, we've initialized action-values to 0.0. However, initializing them with values different from zero can yield varied outcomes.

Optimistic initial values encourage early exploration but may not be suitable for non-stationary problems, where the environment's dynamics change over time.

Determining the appropriate optimistic initial value can also pose a challenge, as it's often unclear what value would be most effective in driving exploration.

4.5.4 UCB action selection

In epsilon-greedy action selection, we had:

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

However, we can improve upon the uniform function with the following approach:

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

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

4.6 Temporal difference learning

Dynamic Programming (DP) necessitates knowledge of the Markov Decision Process (MDP) dynamics. On the other hand, Monte Carlo (MC) learning relies on experience but mandates complete episodes for updating. Consequently, it's solely applicable to episodic tasks. However, even within episodic tasks, MC might encounter challenges.

4.6.1 Temporal-Difference Policy Evaluation with TD(0)

Temporal-Difference combines MC (model-free) with DP (bootstrapping):

$$V(S_t) = V(S_t) + \alpha[G_t - V(S_t)] = V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Here, $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is the Temporal-Difference Error δt.

Algorithm 9 TD(0) Policy Evaluation

```
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}^+$
$V(\text{terminal}) = 0$
for each episode do
    Initialize $S$
    repeat for each step of episode
        $A$ = action given by $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) = V(S) + \alpha[R + \gamma V(S') - V(S)]$
        $S = S'$
    until $S$ is terminal
end for
```

The algorithm takes as input the policy π to be evaluated, and a parameter $\alpha \in (0, 1]$ that represent the step size. One main advantage is that the value function is updated during the episode and not after.

4.6.2 Comparison

Temporal Difference (TD) learning has several advantages over Monte Carlo (MC) learning:

- TD can learn before knowing the final outcome and can update its estimates after every step, whereas MC must wait until the end of an episode before the return is known.
- TD can learn from incomplete sequences, making it more flexible than MC, which can only learn from complete sequences.
- TD is suitable for both continuing (non-terminating) and episodic (terminating) environments, while MC is limited to episodic tasks.
- MC returns are unbiased estimates of the value function, whereas TD targets are biased estimates due to the use of bootstrapping.
- TD targets have lower variance compared to MC returns because they depend on fewer random actions, transitions, and rewards.
- MC works well with function approximation and is less sensitive to initial values, while TD may face challenges with function approximation and is more sensitive to initial values.

Bootstrapping, where updates involve an estimate, is a characteristic of TD and Dynamic Programming (DP), while MC does not bootstrap. Monte Carlo does not rely on Markov assumption. Sampling, where updates do not involve an expected value, is a feature of MC and TD, while DP does not sample.

4.6.3 SARSA

SARSA is an algorithm employed for policy evaluation in reinforcement learning. It operates as an on-policy optimization method, meaning it evaluates and improves the same policy that is used to make decisions. The update rule for SARSA is defined as follows:

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

SARSA is typically paired with an ε-greedy policy for improvement. This means that most of the time, the policy selects the action with the highest estimated value, but occasionally explores other actions with probability ε.

The algorithm requires two parameters: a step size $\alpha \in (0, 1]$ and a small $\varepsilon > 0$.

Algorithm 10 SARSA on-policy control algorithm

```
Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$
$Q(\text{terminal}, \cdot) = 0$
loop
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$
    repeat for each step of episode
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$
        $Q(S, A) = Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$
        $S = S'$
        $A = A'$
    until $S$ is terminal
end loop
```

4.6.4 Q-learning

Q-learning is an algorithm utilized for policy evaluation in reinforcement learning. It operates as an off-policy optimization method, meaning it evaluates a policy while following a different policy for action selection.

As opposed to SARSA, which is a sampled version of the Bellman expectation equation, Q-learning is based on a sampled version of the Bellman optimality equation:

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

The algorithm requires two parameters: a step size $\alpha \in (0, 1]$ and a small $\varepsilon > 0$.

Algorithm 11 Q-learning algorithm

```
Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$
$Q(\text{terminal}, \cdot) = 0$
loop
    Initialize $S$
    repeat for each step of episode
        Choose $A$ from $S$ using policy derived from $Q$
        Take action $A$, observe $R$, $S'$
        $Q(S, A) = Q(S, A) + \alpha\,(R + \gamma \max_a Q(S', a) - Q(S, A))$
        $S = S'$
    until $S$ is terminal
end loop
```

Q-learning updates the Q-values based on the observed rewards and transitions, aiming to find the optimal policy by maximizing the estimated action values over time. Since it's off-policy, it doesn't follow the policy it's evaluating, making it particularly useful in scenarios where exploration and exploitation need to be decoupled.

4.6.5 Eligibility traces

Eligibility traces are a concept in reinforcement learning that play a crucial role in updating the value estimates of states or actions. They are used in combination with temporal difference (TD) learning methods like SARSA (State-Action-Reward-State-Action) or Q-learning. The main features are:

1. *Temporal credit assignment*: eligibility traces help in assigning credit or blame to actions taken in the past for the rewards received in the future.

2. *Memory mechanism*: eligibility traces serve as a memory mechanism that tracks the eligibility of states or actions to be updated based on future rewards. They maintain a record of recent state-action pairs that are likely to contribute to future rewards.

3. *Decay factor*: determines how much past experiences influence the current update. It helps balance between short-term and long-term credit assignment.

4. *Updating value estimates*: when a reward is received, the eligibility traces are used to update the value estimates of relevant states or actions. This updating process is done more efficiently because the traces highlight which experiences are relevant.

5. *Efficiency and learning speed*: By allowing updates to propagate more efficiently through the learning process, eligibility traces can speed up learning and improve the convergence of RL algorithms.

Overall, eligibility traces enhance the efficiency and effectiveness of temporal difference learning methods by providing a mechanism for assigning credit over time, which is crucial for learning in complex environments with delayed rewards.

APPENDIX A

Algebra and statistics

A.1 Least squares

Let's reconsider a dataset consisting of N inputs $\mathbf{x}_i = (x_{i1}, \dots, x_{iD})$, where each $x_{ij} \in \mathbb{R}$ represents a feature with dimension D, along with a target $t_i \in \mathbb{R}$ associated with each input $\mathbf{x}_i$.

Our aim is to predict the target t_i by computing a linear combination of the input $\mathbf{x}_i$, which involves generating a parameter vector $\mathbf{w} = (w_1, \dots, w_D)^T$ that minimizes a certain loss function. Specifically, if we consider the loss function to be the summation of squared prediction errors, it corresponds to Least Square minimization.

Now, let's define the following loss function:

$$L(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{N} \left(t_i + \sum_{j=1}^{D} x_{ij} w_j \right)^2$$

By defining the matrix $X = \begin{bmatrix} \mathbf{x}_1 & \cdots & \mathbf{x}_N \end{bmatrix}^T$, we can rewrite the loss function as:

$$L(\mathbf{w}) = \frac{1}{2} \left\| \mathbf{t} - X\mathbf{w} \right\|_2^2 = \frac{1}{2} (\mathbf{t} - X\mathbf{w})^T (\mathbf{t} - X\mathbf{w})$$

Minimizing the Loss To minimize the loss, we need to compute its derivatives with respect to each component of $\mathbf{w}$:

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \left(\frac{\partial L(\mathbf{w})}{w_1}, \dots, \frac{\partial L(\mathbf{w})}{w_D} \right)$$

In this case, we have two different formulations for the derivative:

1. Traditional (element-wise) derivative:

$$\left(\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} \right)_h = \frac{\partial L(\mathbf{w})}{w_h} = \frac{\partial}{\partial w_h} \left[\frac{1}{2} \sum_{i=1}^{N} \left(t_i - \sum_{j=1}^{D} x_{ij} w_j \right)^2 \right]$$

2. Matrix derivative:

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} \left[\frac{1}{2} (\mathbf{t} - X\mathbf{w})^T (\mathbf{t} - X\mathbf{w}) \right]$$

A.2 Matrices

Eigenvalues and eigenvectors For a square matrix $\mathbb{R}^{n\times n}$, the corresponding eigenvector equations are given by:

$$A\mathbf{v}_i = \lambda_i\mathbf{v}_i$$

Here:

- The eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ represent directions unaffected by the transformation A.
- The eigenvalues $\lambda_1, \ldots, \lambda_n$ determine the scaling factor for the corresponding eigenvectors $\mathbf{v}_i$.

In matrix notation, we can express this relationship as:

$$(A - \lambda I_n)\mathbf{v} = 0$$

This equation has a non-trivial solution only if the rank of the matrix $A - \lambda I_n$ is full, or equivalently:

$$|A - \lambda I_n| = 0$$

Property A.2.1. The rank of A is equal to the number of non-zero eigenvalues.

Property A.2.2. The determinant of A is equal to the product of its eigenvalues:

$$|A| = \prod_{i=1}^{n} \lambda_i$$

Trace The trace of A, denoted as $\mathrm{Tr}(A)$, is equal to the sum of its eigenvalues:

$$\mathrm{Tr}(A) = \sum_{i=1}^{n} \lambda_i$$

A.2.1 Properties

Definition (*Positive definite matrix*)**.** A matrix A is said to be positive definite if $\mathbf{x}^T A\mathbf{x} > 0$ for all vectors $\mathbf{x} \in \mathbb{R}^n \setminus \{0\}$.

A positive definite matrix has all positive eigenvalues, i.e., $\lambda_i > 0$ for all i.

Definition (*Semi-positive definite matrix*)**.** A matrix A is said to be semi-positive definite if $\mathbf{x}^T A\mathbf{x} \geq 0$ for all vectors $\mathbf{x} \in \mathbb{R}^n \setminus \{0\}$.

A semi-positive definite matrix has all non-negative eigenvalues, i.e., $\lambda_i \geq 0$ for all i.

A.3 Random variables

A discrete random variable X is a variable with values in a discrete set, whose value is determined by a stochastic phenomenon. We define a probability function $\mathrm{P} : E \to [0, 1]$ which indicates the likelihood of events in E:

$$\mathrm{P}(X = i) = \frac{|i|}{|E|}$$

A properly defined probability function should satisfy the following properties:

1. $\forall i \in E$, $0 \leq \mathrm{P}(X = 1) \leq 1$.
2. $\sum_{i \in E} \mathrm{P}(X = i) = 1$.

Cumulative function Assuming events are ordered, a cumulative function $F : E \to [0, 1]$ defines the probability of multiple events:

$$F(i) = \mathrm{P}(X \leq 1) = \sum_{h=1}^{i} \mathrm{P}(X = h) = \sum_{h \in E, h \leq i} \frac{|h|}{|E|}$$

A properly defined cumulative function should satisfy the following properties:

- $0 \leq F(i) \leq 1$
- $F(i) = 0$ for all $i < \min_{h \in E} h$
- $F(i) = 1$ for all $igeq \max_{h \in E} h$

Random variables characteristics Quantities characterizing a random variable include the expected value (moment of order one):

$$\mathbb{E}\left[X\right] = \sum_{i \in E} i\mathrm{P}(X = i)$$

And the variance (moment of order two):

$$\mathrm{Var}(X) = \sum_{i \in E} (\mathbb{E}\left[X\right] - i)^2 \mathrm{P}(X = i)$$

The standard deviation of the variance is defined as:

$$\mathrm{std}(X) = \sqrt{\mathrm{Var}(X)}$$

A.3.1 Continuous random variable

Similarly, if the set E is not discrete, we could define the probability density function as:

$$f(x) = \lim_{\delta x \to 0} \frac{\mathrm{P}(x \leq X \leq x + \delta x)}{\delta x}$$

The properties of continuous random variables are:

- $f(x) \geq 0$ for all $x \in \Omega$
- $\int_{x \in \Omega} f(x)\, dx = 1$

Cumulative density function The cumulative density function is defined as:

$$F(x) = \int_{s\in\Omega, s\leq x} f(s)\, ds$$

The cumulative density function has the following properties:

- $0 \leq F(x) \leq 1$ for all $x \in \Omega$
- $F\left(\min_{x\in\Omega} x - \varepsilon\right) = 0$ for all $\varepsilon > 0$
- $F\left(\max x \in \Omega x\right) = 1$

A quantile of order α is defined as a point $z_\alpha \in \Omega$ such that:

$$F(z_\alpha) = 1 - \alpha$$

or a point of the domain leaving to its left a cumulated probability of $1 - \alpha$.

Random variables characteristics Quantities which characterize a random variable are the expected value (moment of order one):

$$\mu = \mathbb{E}\left[X\right] = \int_{x\in\Omega} x f(x)\, dx$$

And the variance (moment of order two):

$$\sigma^2 = \mathrm{Var}(X) = \int_{x\in\Omega} \left(\mathbb{E}\left[X\right] - x\right)^2 f(x)\, dx$$

Gaussian distribution The Gaussian distribution is expressed as:

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

And the relative cumulative density function:

$$F(x, \mu, \sigma) = \int_{-\infty}^{x} f(t, \mu, \sigma)\, dt$$

A.4 Distributions

Usually, we do not have any information about the distribution of random variables. Therefore, we need to estimate their mean and variance. A consistent estimator for the expected value is:

$$\bar{X} = \frac{\sum_{i=1}^{n} x_i}{n}$$

A consistent estimator for the variance is:

$$\bar{s} = \frac{\sum_{i=1}^{n} \left(\bar{X} - x_i\right)^2}{n-1}$$

Theorem A.4.1 (Central limit)**.** *Assuming $\{X_1, \ldots, X_n\}$ as a sequence of independent and identically distributed random variables with $\mathbb{E}\left[X_i\right] = \mu$ and $Var[X_i] = \sigma^2 < \infty$, then:*

$$\sqrt{n}\left(\frac{\sum_{i=1}^{n} X_i}{n} - \mu\right) \to \mathcal{N}(0, \sigma^2)$$

where the convergence holds in distribution.

A.5 Confidence intervals

We need to establish a level of confidence to determine if our estimator is sufficiently accurate. Since the probability of $\bar{X} = \mathbb{E}[X]$ is zero, given that the realization of the expected value is a continuous random variable itself, we need to construct intervals where we have high confidence that the true mean $\mathbb{E}[X]$ lies within. A 95% confidence interval implies that it works correctly 95% of the time.

The available options for confidence intervals are:

- Gaussian approximation:
$$\bar{X} - \frac{z_{\frac{\alpha}{2}}\sigma}{\sqrt{n}} \leq \mu \leq \bar{X} + \frac{z_{\frac{\alpha}{2}}\sigma}{\sqrt{n}}$$

- Chebyshev's inequality (requires $\mathbb{E}[X] = \mu < \infty$ and $\mathrm{Var}[X] = \sigma^2 < \infty$):
$$\bar{X} - \frac{\sigma}{\sqrt{n}\sqrt{\alpha}} \leq \mu \leq \bar{X} + \frac{\sigma}{\sqrt{n}\sqrt{\alpha}}$$

- Hoeffding's inequality (finite support):
$$\bar{X} - (b-a)\sqrt{\frac{-\log(\frac{\alpha}{2})}{2n}} \leq \mu \leq \bar{X} + (b-a)\sqrt{\frac{-\log(\frac{\alpha}{2})}{2n}}$$

A.6 hypothesis testing

In hypothesis testing, we aim to demonstrate that the estimated parameter $\bar{X}$ is equal to μ and that another estimated parameter $\bar{X}'$ is different from yet another estimated parameter $\bar{X}''$. With statistics, we define a null hypothesis H_0 and an alternative hypothesis H_1:

$$H_0 : \mu = \mu_0 \qquad \text{vs} \qquad H_1 : \mu \neq \mu_0$$

and utilize the data to provide evidence supporting either hypothesis.

A.6.1 Basic Gaussian test

Given the data $\{x_1, \ldots, x_n\}$, we have:

$$\bar{X} \sim \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right) \to t = \frac{\bar{X} - \mu}{\frac{\sigma}{\sqrt{n}}} \sim \mathcal{N}(0,1)$$

Fixing a confidence $1 - \alpha$ with $\alpha \in (0,1)$, the test statistic t should be close to the true mean μ with very high probability. Formally:

$$\mathrm{P}(t < z_{\frac{\alpha}{2}} \lor t > z_{1-\frac{\alpha}{2}}) = \alpha$$

The corresponding decision table is:

		Decision	
		Fail to reject $\mathbf{H_0}$	**Reject $\mathbf{H_0}$**
True	$\mathbf{H_0}$	Correct	Type I error (α)
	$\mathbf{H_1}$	Type II error	Correct

A.6.2 P-value

To avoid specifying the confidence α, we can let the data inform us about how confident we might be about their correspondence to a specific hypothesis:

- Small p-values imply that we are confident that the H_1 hypothesis holds.
- Large p-values imply that we are not able to reject the H_0 hypothesis.

A.7 Bayesian approach

The bounds provided by traditional methods do not allow for the incorporation of information one has about the distribution parameters. A new approach considers the expected value μ of the random variable X as a random variable itself. Bayes' formula is utilized to update this information:

$$\mathrm{P}(a|b) = \frac{\mathrm{P}(b|a)\mathrm{P}(a)}{\mathrm{P}(b)}$$

Considering Bayes' formula, we have:

$$\begin{aligned}
\mathrm{P}(\mu|x_1,\ldots,x_t) &= \frac{\mathrm{P}(x_1,\ldots,x_t|\mu)\mathrm{P}(\mu)}{\mathrm{P}(x_1,\ldots,x_t)} \\
&\propto \mathrm{P}(x_t|\mu)\mathrm{P}(x_1,\ldots,x_{t-1}|\mu)\mathrm{P}(\mu) \\
&= \mathrm{P}(x_t|\mu)\mathrm{P}(x_{t-1}|\mu)\mathrm{P}(x_1,\ldots,x_{t-2}|\mu)\mathrm{P}(\mu) \\
&= \mathrm{P}(\mu)\prod_{h=1}^{t}\mathrm{P}(x_h|\mu)
\end{aligned}$$

We incrementally incorporate information from a prior distribution $\mathrm{P}(\mu)$.