Machine Learning

An Unofficial Guide to the Georgia Institute of Technology's CS7641

Christian Salas
csalas9@gatech.edu

Last Updated: May 23, 2025

These notes were created with personal effort and dedication. They may contain typos, errors, or incomplete sections. If you find them helpful or wish to provide feedback, feel free to reach out.

Have fun!

Disclaimer: I am a student, not an expert. While I aim for accuracy, there may be mistakes. If you spot any issues, please contribute or contact me.

Contents

1 Introduction

Machine Learning (ML) is a subset of artificial intelligence that enables systems **to learn and improve from experience without being explicitly programmed.** It involves developing algorithms that can make decisions or predictions based on data.

1.1 Types of Machine Learning

1.1.1 Supervised Learning

Supervised learning **involves training a model using labeled data**, where the input-output pairs are provided. The goal is to learn a mapping from inputs to outputs.

Examples: Classification, Regression
Algorithms: Support Vector Machines (SVM), Decision Trees, Neural Networks

1.1.2 Unsupervised Learning

Unsupervised learning **deals with data without labeled outputs**. The system attempts to find patterns or structures in the data.

Examples: Clustering, Dimensionality Reduction
Algorithms: K-Means, Principal Component Analysis (PCA), Autoencoders

1.1.3 Reinforcement Learning

Reinforcement learning is about **learning through interactions with an environment**. The system learns by receiving **rewards** or **penalties** based on its actions.

Examples: Game Playing, Robotics
Algorithms: Q-Learning, Deep Q-Networks (DQN)

1.1.4 Comparison Table

Aspect	**Supervised Learning**	**Unsupervised Learning**	**Reinforcement Learning**
Data Requirement	Labeled data	Unlabeled data	Environment interaction
Goal	Predict output	Find hidden patterns	Maximize rewards
Common Tasks	Classification, Regression	Clustering, Anomaly Detection	Game Playing, Robotics
Feedback Type	Direct (labeled outputs)	Indirect (no labels)	Reward-based
Algorithms	SVM, Neural Networks	K-Means, PCA	Q-Learning, DQN

Table 1: Comparison of Machine Learning Types

2 Supervised Learning

> *"More data beats clever algorithms, but better data beats more data."*
> – Andrew Ng

Definition: Supervised learning is a type of machine learning where a model **learns from labeled data** maps inputs X to outputs Y. It aims to learn a function $f : X \to Y$ by minimizing prediction errors. Tasks include:

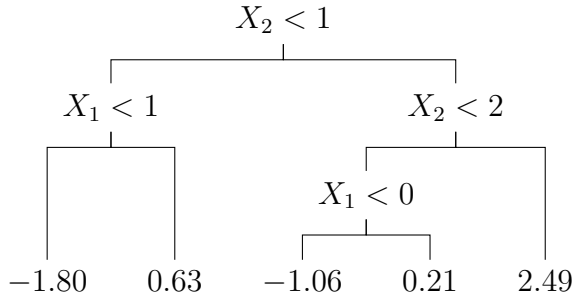
1. **Classification:** Predicting categorical labels (e.g., image detection).
2. **Regression:** Predicting continuous values (e.g., house prices).

2.1 Classification Learning

Classification learning is a type of supervised learning where the goal is to **assign input data (instances) to predefined categories or labels based on learned patterns from training data**

1. **Instances (Input):** The data points or examples used as input to the classification model.
2. **Concept Function:** A mapping function that assigns a class label (e.g., true/false) to each instance based on its features. **This function represents the actual rule or decision boundary.**
3. **Target Concept:** The ideal function that **correctly classifies all possible instances**. It is what the learning algorithm tries to approximate. **(actual answer)**
4. **Hypothesis:** A **potential** function generated by the learning algorithm that approximates the target concept based on the training data.
5. **Sample (Training Data):** A collection of labeled instances used by the learning algorithm to build the classification model.
6. **Candidate Hypothesis:** A possible hypothesis that the learning algorithm considers during the learning process. The best candidate is selected based on its performance on the training data.
7. **Testing Set:** A separate set of labeled instances used to evaluate the performance and generalization ability of the trained model.

2.2 Decision Trees



Decision trees are a simple yet powerful method for decision-making and data classification. They work by recursively splitting the data based on the best attribute, asking questions, and following paths to arrive at a conclusion.

How Decision Trees Work

The process of building a decision tree can be summarized as follows:

1. **Pick the Best Attribute:** The "best" attribute is the one that splits the data most effectively, often aiming to evenly divide the data. Common metrics **include information gain or Gini impurity**.
2. **Ask a Question:** Formulate a question based on the chosen attribute (e.g., "Is the value greater than X?").
3. **Follow the Answer Path:** Depending on the answer (e.g., Yes/No), follow the corresponding branch of the tree.
4. **Repeat Until a Conclusion:** Continue this process recursively until the data is fully classified or a specific answer is reached.

Expressiveness of Decision Trees

Decision trees are highly expressive and can represent a variety of logical expressions, such as:

- **AND:** Requires all conditions to be true.
- **OR:** Requires at least one condition to be true.
- **XOR:** Requires exactly one condition to be true (either-or, but not both).

Power of Expressiveness:

- Decision trees can represent even highly complex relationships, including exponential structures like XOR.
- For n attributes (assuming boolean values), the number of possible decision trees grows exponentially.

2.3 ID3 Algorithm

The **Iterative Dichotomiser 3 (ID3)** algorithm is a foundational method for constructing decision trees. It uses a greedy approach to build a tree by selecting attributes that maximize information gain at each step.

1. **Start with the Root:** Treat the entire dataset as the root node.
2. **Choose the Best Attribute:** For each attribute, calculate the **information gain** based on entropy. The attribute with the **highest** information gain is selected as the splitting criterion.
3. **Split the Data:** Partition the dataset into subsets based on the values of the chosen attribute.
4. **Repeat Recursively:** For each subset, repeat the process of choosing the best attribute and splitting, creating child nodes. Continue until:
   - All instances in a subset belong to the same class (pure node).
   - There are no remaining attributes to split.
   - A stopping criterion (e.g., maximum depth) is met.
5. **Assign a Class Label:** For any remaining nodes where further splitting isn't possible, assign the most frequent class label in that subset.

Core Concepts in ID3

- **Entropy: A measure of randomness** in the dataset. Lower entropy implies higher purity.
- **Information Gain:** The reduction in entropy after a dataset is split on an attribute. It is defined as:

$$\text{Information Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v)$$

  where S is the dataset, A is the attribute, and S_v represents the subset of S for a specific value v of A.

Overfitting in Decision Trees

Overfitting occurs when a decision tree becomes too complex, learning not only the underlying patterns in the training data but also the noise and randomness. This results in a model that performs well on the training set but fails to generalize to unseen data.

Causes of Overfitting

Decision trees are particularly prone to overfitting due to:

- **Excessive Depth:** Splitting the tree too many times can lead to nodes with very few samples, which capture noise rather than meaningful patterns.

- **Small Subsets:** When the data is divided into increasingly smaller subsets, the tree may overfit to outliers or specific cases.
- **Complex Splits:** Using attributes that create overly specific splits can reduce generalizability.

Symptoms of Overfitting

A decision tree may be overfitting if:

- **High Training Accuracy, Low Test Accuracy:** The model performs perfectly on the training data but poorly on validation or test data.
- **Overly Complex Tree:** The tree has too many branches or splits, making it difficult to interpret.

Strategies to Avoid Overfitting

To prevent overfitting, several strategies can be employed:

1. **Pruning:**
    - **Pre-pruning:** Limit the maximum depth of the tree or require a minimum number of samples per node to stop splits early.
    - **Post-pruning:** Simplify the tree after it is fully grown by removing branches that do not significantly improve accuracy.
2. **Regularization:** Use techniques such as:
    - Limiting the number of splits or leaf nodes.
    - Adding penalties for tree complexity in the objective function.
3. **Cross-Validation:** Use cross-validation to evaluate the model's performance and select a tree size that minimizes validation error.
4. **Use Ensemble Methods:** Techniques like Random Forests or Gradient Boosted Trees combine multiple trees and inherently reduce the risk of overfitting.

2.4 Regression

Regression is a statistical method used to model and analyze the relationships between variables. In the context of machine learning, it is primarily concerned with predicting a continuous output given a set of input features.

2.4.1 Function Approximation and Its Relationship to Regression

Function approximation involves estimating an unknown function $f(x)$ using a known model based on given data. Regression is a key method for function approximation, as it focuses on finding the best function (or line, curve, etc.) that minimizes the error between the predicted values and the actual observed data.

Regression in Machine Learning In machine learning, regression aims to find the best fit for data by minimizing a loss function, such as the sum of squared errors. Techniques to achieve this include:

- **Hill Climbing:** Iteratively adjusting parameters to maximize or minimize an objective function.
- **Calculus:** Using derivatives (e.g., gradient descent) to find the minimum of the loss function.
- **Random Search:** Randomly sampling parameter values to identify the best fit.

Finding the Best Function (Line) Regression minimizes a loss function, such as the squared error:

$$E(c) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where y_i is the observed value, $\hat{y}_i$ is the predicted value, and n is the number of data points. Various types of errors, such as absolute error or squared error, can be used depending on the application.

Example: Given x, y Data Points Consider a dataset of x and y values. We can fit a polynomial function to this data:

$$f(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_k x^k$$

What is c? In the polynomial $f(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_k x^k$, the c_i terms are the coefficients. They determine the contribution of each term:

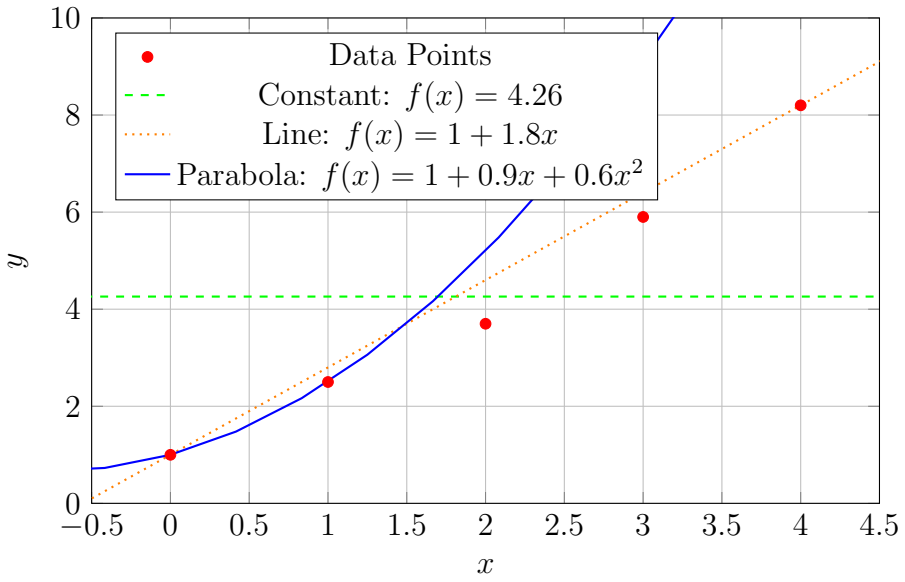
- c_0: Intercept, the value of $f(x)$ when $x = 0$.
- c_1: Coefficient of x, controls the slope (linear term).
- c_2: Coefficient of x^2, controls the curvature (quadratic term).
- c_k: Coefficients for higher-order terms, affecting complex shapes.

These coefficients are calculated to best fit the given data points **and** k is the order of the polynomial:

- $k = 0$: Constant
- $k = 1$: Line
- $k = 2$: Parabola

2.4.2 Visualizing the Fit

We choose the line because it aligns more closely with our data while maintaining simplicity. The degree of the model determines how closely it adheres to the data points without overfitting.



2.5 Cross Validation

2.5.1 Training and Test Sets

When building a model, we start with a dataset (training set) to learn patterns. For example, we might attempt to fit a line to the data. If the line doesn't fit well, we could consider a higher-order polynomial.
However, increasing complexity risks overfitting, where the model performs well on the training set but poorly on the test set. The test set is never used during training. It serves as a stand-in for future, unseen data, ensuring that the model generalizes well. This is critical for assessing the model's ability to perform in real-world scenarios.

2.5.2 Fundamental Assumption: IID Data

In supervised learning (SL), we rely on the fundamental assumption that all data is independently and identically distributed (IID). This means:

- All data (training, validation, and test sets) is drawn from the same source.
- Performance on the test set reflects how the model will perform on unseen data in the future.

If this assumption is violated, conclusions drawn from the model's performance may not be valid.

2.5.3 Using a Validation Set

To avoid overfitting and select the best model, we introduce a validation set. This is created by taking a percentage of the training data and setting it aside. The steps are as follows:

1. Split the training data into two parts:
    - A smaller subset for validation (the cross-validation set).
    - The remaining data for training.
2. Train the model on the training subset.
3. Evaluate its performance on the validation set.

This ensures that we don't overfit by relying only on the training set.

2.5.4 K-Fold Cross Validation

K-fold cross validation is a systematic way to use all data for validation without wasting it. The process is as follows:

1. Split the training data into k equal-sized folds.
2. For each fold:
    - Use $k-1$ folds for training.
    - Use the remaining fold for validation.
3. Average the validation errors across all folds to estimate model performance.

The model with the lowest average error across folds is selected as the best model.

2.5.5 Balancing Model Complexity and Generalization

The goal of cross-validation is to find a model that is:

- Complex enough to fit the training data accurately.
- Simple enough to avoid poor performance on the validation and test sets.

This balance helps ensure that the model generalizes well to unseen data.

2.6 Other Input Spaces

2.6.1 Scalar Input, Continuous

A scalar input is a single continuous value, such as temperature, speed, or time. In regression or classification, the goal is to model the relationship between this single input and the target output:

$$y = f(x)$$

For example, a simple linear regression model might predict y (house price) based on x (square footage). Scalar inputs are the simplest input space and are often used in introductory examples.

2.6.2 Vector Input, Continuous

Vector inputs consist of multiple continuous features, represented as:

$$\mathbf{x} = [x_1, x_2, \ldots, x_n]$$

Each feature x_i contributes to predicting the output. The model's task is to learn a function that maps the input vector $\mathbf{x}$ to the output y:

$$y = f(\mathbf{x})$$

For example, in predicting a car's price, the input vector might include features such as mileage, engine size, and year of manufacture. Continuous vector inputs allow models to capture complex relationships across multiple dimensions.

2.6.3 Hyperplanes in Input Spaces

Hyperplanes are geometric constructs that generalize lines (in 2D) and planes (in 3D) to higher-dimensional spaces. In machine learning, hyperplanes are often used as decision boundaries:

$$\mathbf{w}^T\mathbf{x} + b = 0$$

where:

- $\mathbf{w}$: Weight vector defining the orientation of the hyperplane.
- b: Bias term shifting the hyperplane.
- $\mathbf{x}$: Input vector.

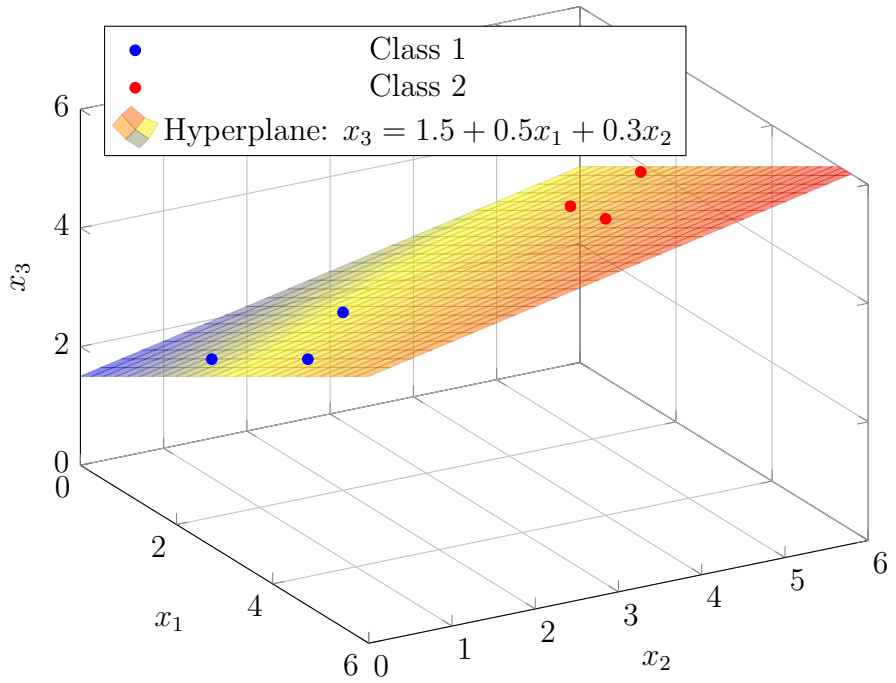
Hyperplanes partition the input space into regions, such as:

$$\mathbf{w}^T\mathbf{x} + b > 0 \quad \text{(one class)}$$

$$\mathbf{w}^T\mathbf{x} + b < 0 \quad \text{(another class)}$$

For example, in a binary classification task with two features (x_1, x_2), the hyperplane might separate data points from two different classes. In higher dimensions, hyperplanes extend this idea to separate complex, multidimensional datasets.

Visual Example: Below is a visualization of a hyperplane separating two classes in a 3D vector space:



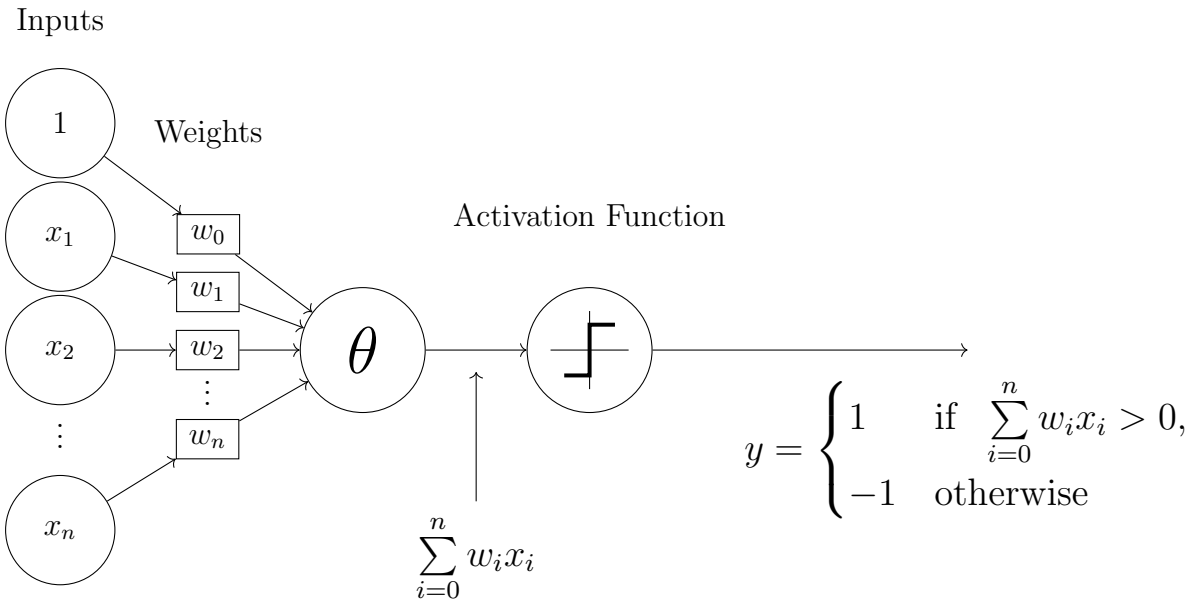
2.7 Neural Networks

Introduction Neural networks are computational models inspired by the structure and function of the human brain. In our brains, a vast network of interconnected neurons processes information, enabling us to perform complex tasks like perception, decision-making, and learning. Similarly, artificial neural networks aim to mimic this biological process by using interconnected nodes (artificial neurons) to process data and learn patterns.

Biological neural networks consist of:

- **Neurons**: The fundamental units of the brain that receive, process, and transmit information.
- **Synapses**: Connections between neurons through which signals are passed.
- **Axons and Dendrites**: Structures that transmit signals between neurons.

2.7.1 Artificial Neural Networks



Where:

- x_i: The i-th input feature (e.g., $x_1, x_2, \ldots, x_n$).
- w_i: The weight associated with the i-th input feature, indicating its contribution to the output.
- θ: The theta node (firing threshold), which computes $\sum_{i=0}^{n} w_i x_i$, the weighted sum of the inputs.
- *Activation Function*: Applies a non-linear transformation to the weighted sum, introducing non-linearity into the model.
- *y (output)*: The final prediction of the perceptron, based on the activation function's result:

$$output = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0, \\ -1 & \text{otherwise.} \end{cases}$$

2.7.2 How Powerful is a Perceptron Unit?

A perceptron is a simple linear classifier that determines a decision boundary based on its weights and bias. The decision boundary separates the input space into two regions corresponding to different output classes (0 or 1). This section illustrates how the perceptron determines the decision boundary and classifies points for basic logical operations: AND, OR, and NOT.

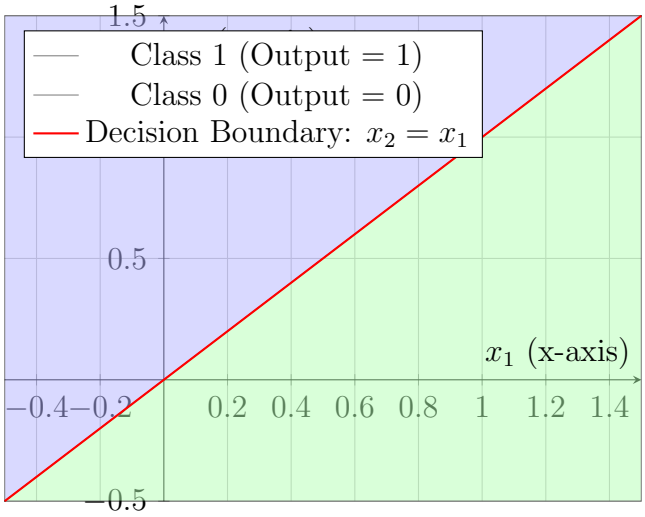


Figure 1: Perceptron Decision Boundary for Logical Operations.

Examples of Logical Operations

AND Gate:

- Inputs: $x_1, x_2 \in \{0, 1\}$.
- Output: $y = 1$ if $x_1 = 1$ and $x_2 = 1$; otherwise, $y = 0$.
- Perceptron parameters:

$$w_1 = 1, \quad w_2 = 1, \quad \theta = 1.5.$$

- Decision boundary: $x_2 = -x_1 + 1.5$.

OR Gate:

- Inputs: $x_1, x_2 \in \{0, 1\}$.
- Output: $y = 1$ if $x_1 = 1$ or $x_2 = 1$; otherwise, $y = 0$.
- Perceptron parameters:

$$w_1 = 1, \quad w_2 = 1, \quad \theta = 0.5.$$

- Decision boundary: $x_2 = -x_1 + 0.5$.

NOT Gate:

- Input: $x_1 \in \{0, 1\}$.
- Output: $y = 1$ if $x_1 = 0$; otherwise, $y = 0$.
- Perceptron parameters:

$$w_1 = -1, \quad \theta = -0.5.$$

- Decision boundary: $x_1 = 0.5$.

Key Takeaways:

1. The perceptron can learn any linearly separable function, such as AND, OR, and NOT gates.
2. Non-linear functions like XOR cannot be learned by a single perceptron, as their decision boundary is not linear.

2.7.3 Perception Training

The goal of perceptron training is to **find a set of weights that map inputs to their desired outputs**. This is done by iteratively adjusting the weights based on the error between the perceptron's predicted output and the actual output for a given example.

Training Algorithm: The perceptron training rule is:

$$w_i \leftarrow w_i + \Delta w_i,$$

where:

$$\Delta w_i = \eta \cdot (y - \hat{y}) \cdot x_i,$$

$$\hat{y} = \text{activation}\left(\sum_{i=1}^{n} w_i x_i - \theta\right) \geq 0,$$

Here:

1. w_i: The weight for input x_i,
2. η: The learning rate, controlling the step size for weight updates,
3. y: The desired output (actual label),
4. $\hat{y}$: The perceptron's predicted output,
5. x_i: The input feature corresponding to w_i.

Implementation

To implement perceptron training, we **iteratively update the weights using the training algorithm while there are errors in the model's predictions**. The training process involves the following steps:

Steps:

1. **Initialization:** Initialize the weights w_i (including the bias term $-\theta$) to small random values or zeros. Choose a learning rate η.
2. **Iteration through the training data:**
   (a) For each example $(\mathbf{x}, y)$ in the training dataset:

i. Compute the perceptron's predicted output $\hat{y}$:

$$\hat{y} = \text{activation}\left(\sum_{i=1}^{n} w_i x_i - \theta\right),$$

where activation(z) outputs 1 if $z \geq 0$ and 0 otherwise.

ii. Compare $\hat{y}$ with the actual output y:

$$\text{error} = y - \hat{y}.$$

iii. If there is an error (error $\neq 0$), update the weights:

$$w_i \leftarrow w_i + \eta \cdot \text{error} \cdot x_i \quad \forall i.$$

3. **Stopping Criteria:** Continue training until one of the following conditions is met:

   (a) **No Errors in the Dataset:** All predictions $\hat{y}$ match the actual labels y across the entire training set. This indicates that the perceptron has **converged** to a set of weights that correctly classify the training data.

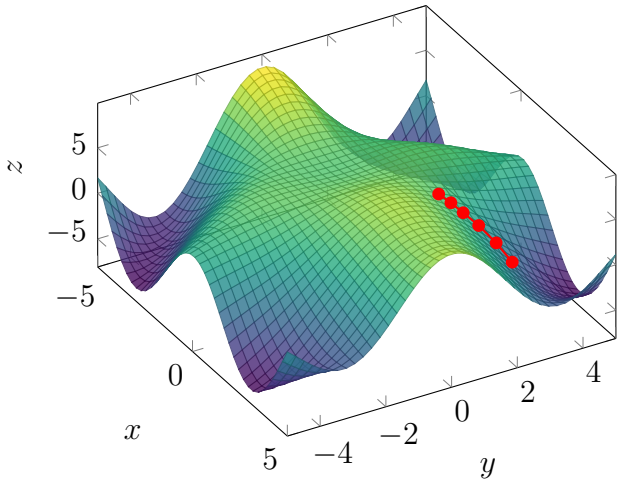
   (b) **Maximum Iterations Reached:** To avoid infinite loops in case of non-linearly separable data, set a maximum number of iterations (epochs). If the perceptron does not converge within this limit, terminate training and consider using additional techniques (e.g., kernel methods or other algorithms).

4. **Output:** Once training stops, the final weights can be used for predictions on new data.

Key Points:

- Perceptron training guarantees convergence only if the data is **linearly separable**. For non-linearly separable data, the perceptron might not find a solution.
- Choosing an appropriate learning rate η is important to ensure smooth and efficient weight updates.
- If convergence issues arise, consider augmenting the dataset or using more advanced models, such as multi-layer perceptrons.

2.7.4 Gradient Descent



The goal of gradient descent is to minimize a given loss function by iteratively updating the model parameters (weights) in the direction of the negative gradient. This optimization technique is widely used in machine learning for training models.

"Gradient", the direction of **steepest** increase. Steps can be summarized as compute the gradient, take a small step in the gradient direction, and repeat.

Algorithm: The gradient descent rule is:

$$w_i \leftarrow w_i - \eta \cdot \frac{\partial L}{\partial w_i},$$

where:

1. w_i: The weight for input x_i,
2. η: The learning rate, controlling the step size for updates,
3. L: The loss function, quantifying the error between predicted and actual outputs,
4. $\frac{\partial L}{\partial w_i}$: The gradient of the loss function with respect to w_i, indicating the direction of steepest ascent.

Implementation

To implement gradient descent, we **iteratively update the weights by calculating the gradient of the loss function and moving in the negative direction of the gradient**. The training process involves the following steps:

Steps:

1. **Initialization:** Initialize the weights w_i to small random values or zeros. Choose a learning rate η.
2. **Iteration through the training data:**
   (a) Compute the loss L for the current set of weights. For example, in linear regression, the loss function could be the Mean Squared Error (MSE):

$$L = \frac{1}{m} \sum_{j=1}^{m} (y_j - \hat{y}_j)^2,$$

   where m is the number of training examples, y_j is the actual output, and $\hat{y}_j$ is the predicted output.

   (b) Calculate the gradient of the loss function with respect to each weight:

$$\frac{\partial L}{\partial w_i} = -\frac{2}{m} \sum_{j=1}^{m} (y_j - \hat{y}_j)\, x_{i,j}.$$

   (c) Update each weight using the gradient descent rule:

$$w_i \leftarrow w_i - \eta \cdot \frac{\partial L}{\partial w_i}.$$

3. **Stopping Criteria:** Continue updating the weights until one of the following conditions is met:

   (a) **Convergence:** The change in the loss L between iterations is below a predefined threshold, indicating that the model has converged to a minimum.

   (b) **Maximum Iterations Reached:** To avoid excessive computation, set a maximum number of iterations. If the gradient descent does not converge within this limit, terminate training and adjust parameters (e.g., learning rate or model complexity).

4. **Output:** Once training stops, the final weights minimize the loss function and can be used for predictions on new data.

Key Points:

- Gradient descent can converge to a local or global minimum, depending on the loss function and the starting weights.
- Choosing an appropriate learning rate η is critical:
  - A learning rate that is too large may cause the algorithm to diverge.
  - A learning rate that is too small may result in slow convergence.
- Variants such as Stochastic Gradient Descent (SGD) and Mini-batch Gradient Descent can be used to improve training efficiency and generalization.

2.7.5 Comparison of Perceptron and Gradient Descent

Both perceptron training and gradient descent are methods used to optimize weights in machine learning models. However, they differ significantly in their approach and applicability.

Key Differences:

1. **Loss Function:**
   - **Perceptron** does not use a continuous loss function but relies on a step function for classification. The error is based on misclassification.
   - **Gradient descent** minimizes a continuous and differentiable loss function (e.g., Mean Squared Error for regression tasks), allowing smooth optimization.
2. **Convergence:**
   - **Perceptron** training guarantees convergence only if the data is linearly separable.
   - **Gradient descent** can optimize models even for non-linearly separable data and often converges to a minimum (local or global).
3. **Updates:**

- **Perceptron** updates weights only for misclassified examples, leading to discrete changes.
- **Gradient descent** updates weights for all examples (batch gradient descent) based on the gradient of the loss function, enabling smoother adjustments.

4. **Applicability:**

   - **Perceptron** is primarily suited for binary classification tasks with linearly separable data.
   - **Gradient descent** is versatile and can be used for a wide range of tasks, including classification, regression, and deep learning.

Visual Comparison:

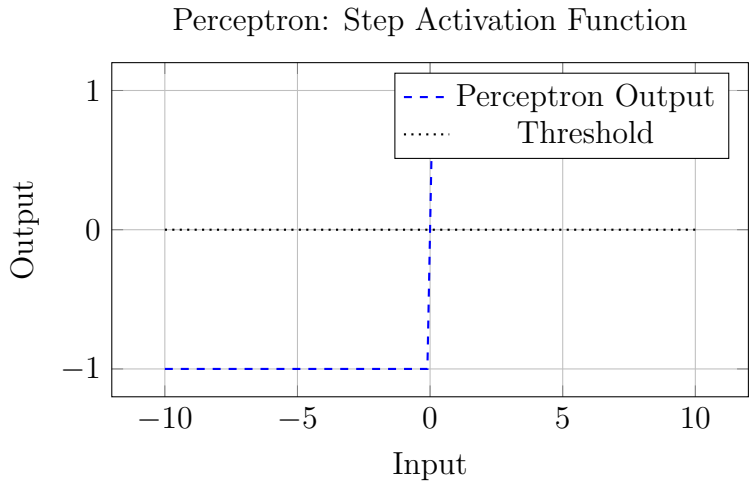


Figure 2: Perceptron: Step Activation Function. The perceptron makes discrete predictions based on a threshold.

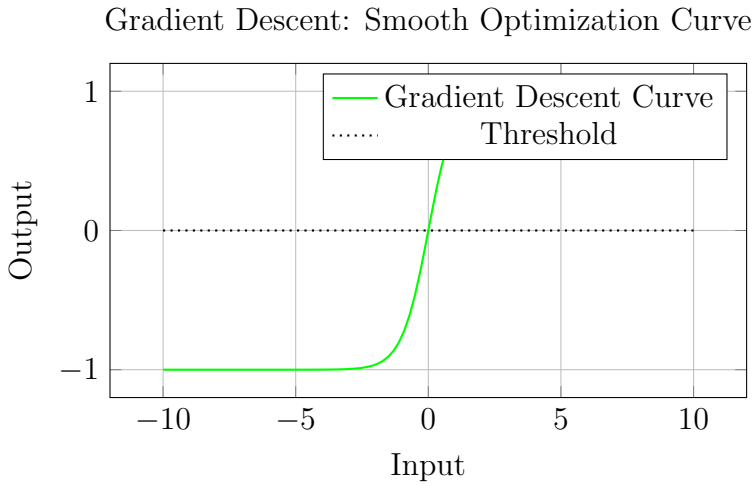


Figure 3: Gradient Descent: Smooth Optimization Curve. Gradient descent iteratively adjusts weights for smooth convergence.

Key Insights:

- Perceptron training is simpler but limited to linearly separable data.
- Gradient descent offers greater flexibility and is a fundamental technique for training more complex models, such as neural networks.

2.7.6 Sigmoid

The sigmoid activation function is widely used in machine learning for mapping inputs to a value between 0 and 1. It is especially common in logistic regression and neural networks for binary classification tasks.

Definition: The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

where:

1. x: The input to the function (often the weighted sum of inputs in a neural network),
2. $\sigma(x)$: The output, which is a value between 0 and 1.

Implementation

To use the sigmoid function in machine learning, we apply it to the model's weighted sum of inputs to compute probabilities or normalized outputs. The implementation involves the following steps:

Steps:

1. **Initialization:** Initialize the weights w_i to small random values or zeros. Set up the input data $\mathbf{x}$ and the bias term θ.
2. **Forward Pass:**
   (a) Compute the weighted sum of inputs:
   $$z = \sum_{i=1}^{n} w_i x_i - \theta.$$
   (b) Apply the sigmoid function to calculate the output:
   $$\sigma(z) = \frac{1}{1 + e^{-z}}.$$
3. **Backpropagation:**
   (a) Compute the gradient of the loss function with respect to the sigmoid output:
   $$\frac{\partial L}{\partial \sigma(z)}.$$

(b) Compute the derivative of the sigmoid function for backpropagation:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

(c) Update the weights using the gradient descent rule:

$$w_i \leftarrow w_i - \eta \cdot \frac{\partial L}{\partial w_i}.$$

4. **Output:** After applying the sigmoid function, the output is a probability or a normalized value between 0 and 1.

Key Points:

- The sigmoid function is smooth and differentiable, making it suitable for optimization using gradient descent.
- The output range $[0, 1]$ is ideal for binary classification tasks, where the value can be interpreted as a probability.
- A downside of the sigmoid function is the "vanishing gradient" problem for very large or small input values x, as $\sigma'(x)$ approaches zero.
- Alternatives such as the ReLU or tanh activation functions are often preferred in modern deep learning models to address the vanishing gradient issue.

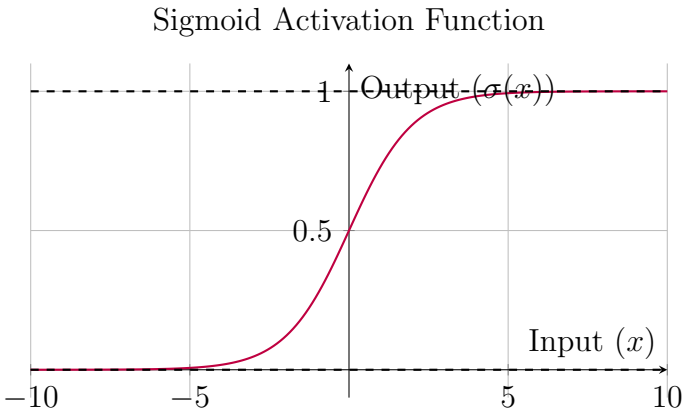
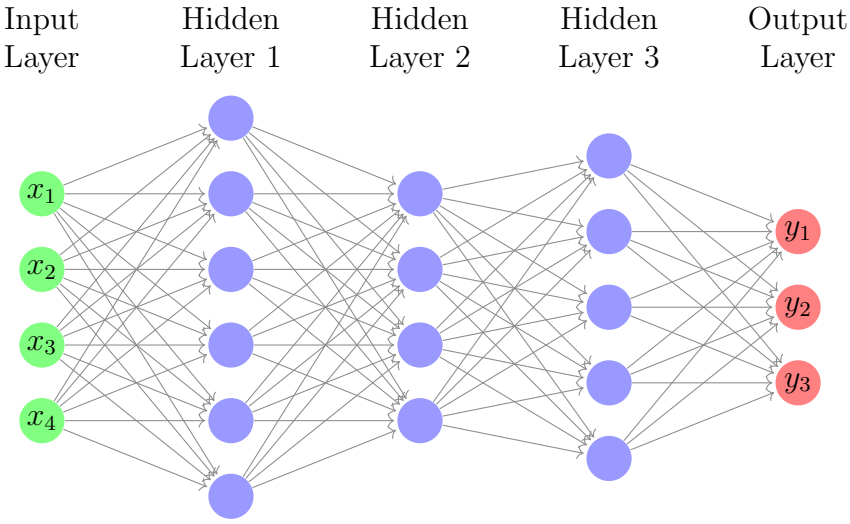


Figure 4: Sigmoid Activation Function. The sigmoid maps inputs to a smooth, continuous range between 0 and 1.

2.7.7 Network Definition



- **Input Layer:** The first layer where data enters the network. Each node corresponds to an input feature, e.g., $x_1, x_2, \ldots, x_n$.
- **Hidden Layers:** Intermediate layers that process inputs through weights and activation functions.
  - **Layer 1:** Initial transformation of input features.
  - **Layer 2:** Deeper feature extraction.
  - **Layer 3:** High-level feature representations.
- **Output Layer:** Final layer producing outputs, $y_1, y_2, \ldots, y_m$. For classification tasks, nodes represent class probabilities.

2.7.8 Backpropagation in the Network

Backpropagation trains the network by minimizing the error between predicted and actual outputs. It involves two key steps:

- **Forward Pass:**
  - Input data flows through the network, layer by layer.
  - Each layer computes a weighted sum of inputs, applies an activation function, and passes the output forward.
- **Backward Pass:**
  - Compute the error between predictions and targets using a loss function (e.g., Mean Squared Error or Cross-Entropy Loss).

- Propagate the error backward through the network, layer by layer, using the chain rule to calculate gradients:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial a_i} \cdot \frac{\partial a_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_{ij}},$$

where L is the loss, a_i is the activation, z_i is the weighted sum, and w_{ij} is the weight.

- **Weight Updates:**
  - Update weights using gradient descent or its variants:

$$w_{ij} \leftarrow w_{ij} - \eta \cdot \frac{\partial L}{\partial w_{ij}},$$

where η is the learning rate.

  - Updates occur per training example (stochastic gradient descent) or per batch (mini-batch gradient descent).
- **Efficiency:**
  - Backpropagation reuses computations from the forward pass, reducing redundancy and improving computational efficiency.
  - Weight updates improve the network's ability to predict in the forward pass.

2.7.9 Restraint and Preference Bias

In neural networks, restraint and preference bias influence the learning process and model outcomes:

- **Restraint Bias:**
  - Imposed constraints that limit the model's ability to overfit or explore certain solutions.
  - Examples include regularization techniques (e.g., L1/L2 regularization) and architectural limitations (e.g., fixed layer sizes).
  - Restraint promotes generalization by discouraging overly complex or specific patterns that may not generalize to new data.
- **Preference Bias:**
  - Encoded tendencies in the network design or training process that prioritize certain solutions over others.
  - Examples include activation function choices, weight initialization strategies, or biases in the training data distribution.
  - Preference bias affects the search for solutions, guiding the network toward outcomes aligned with the imposed biases.
- **Key Insights:**

- Both restraint and preference bias are integral to effective model training, balancing the trade-off between underfitting and overfitting.
- These biases align with the principles of inductive bias, where assumptions guide learning in the presence of limited data.
- Careful calibration of these biases is crucial to ensure the model captures meaningful patterns without being misled by spurious correlations.

2.8 Instance Based Learning

Instance-based learning is a family of algorithms that r**elies on storing and using specific instances from the training data to make predictions, rather than constructing a general model.** These methods defer computation until prediction time, earning them the name *lazy learners*.

A prominent example is the k-Nearest Neighbors (k-NN) algorithm, where predictions for a query point are made based on the similarity to its k closest neighbors in the training set.

2.8.1 k-Nearest Neighbors

The **k-Nearest Neighbors (k-NN)** algorithm is an instance-based learning method that makes predictions by analyzing the k closest points in the training dataset to a query point. Let the training dataset be represented as $\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid i = 1, 2, \ldots, N\}$, where $\mathbf{x}_i \in \mathbb{R}^d$ are feature vectors and $y_i \in \mathcal{Y}$ are their corresponding labels.
Distance Metric: The similarity between a query point $\mathbf{x}_q \in \mathbb{R}^d$ and a training point $\mathbf{x}_i$ is typically measured using a distance function $d(\mathbf{x}_q, \mathbf{x}_i)$. A common choice is the Euclidean distance:

$$d(\mathbf{x}_q, \mathbf{x}_i) = \sqrt{\sum_{j=1}^{d} (x_{qj} - x_{ij})^2}.$$

Prediction:

1. **Classification:** The predicted label $\hat{y}_q$ for the query point is determined by majority voting among the k nearest neighbors:

$$\hat{y}_q = \arg\max_{y \in \mathcal{Y}} \sum_{i \in \mathcal{N}_k(\mathbf{x}_q)} \mathbb{I}(y_i = y),$$

   where $\mathcal{N}_k(\mathbf{x}_q)$ represents the indices of the k nearest neighbors, and $\mathbb{I}$ is the indicator function.

2. **Regression:** The predicted value $\hat{y}_q$ is the average of the outputs of the k nearest neighbors:

$$\hat{y}_q = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x}_q)} y_i.$$

Preference Bias: The choice of the distance metric and the value of k introduces a *preference bias* in k-NN.

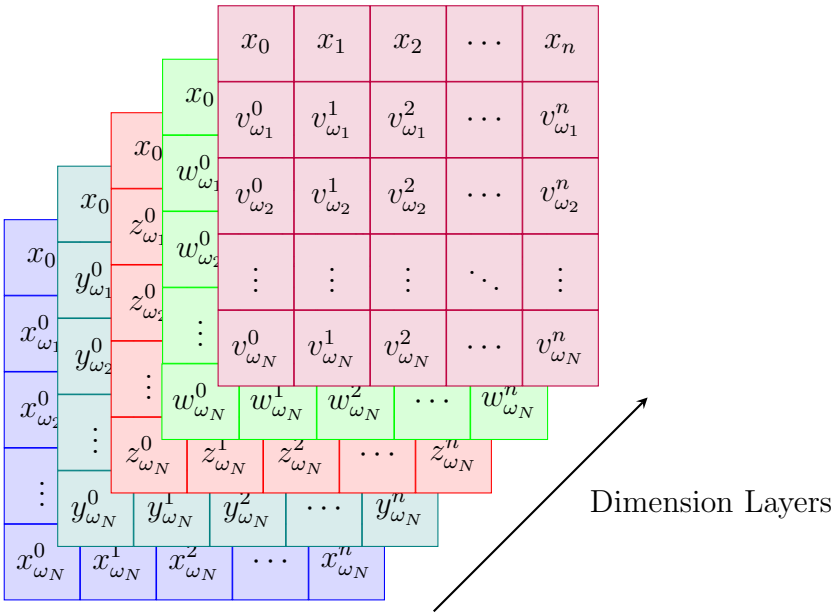
1. **Distance Metric:** Different metrics (e.g., Manhattan, cosine similarity) prioritize different aspects of similarity. For instance, Euclidean distance emphasizes absolute geometric proximity, which may not be ideal for high-dimensional or sparse data.

2. **Choice of k:** A small k can lead to overfitting, as predictions are overly influenced by individual neighbors, reflecting local noise. Conversely, a large k smooths the decision boundary but may overlook subtle patterns in the data, leading to underfitting.

3. **Feature Scaling:** Features with larger numerical ranges dominate the distance calculation unless data is properly normalized, potentially skewing the algorithm's bias.

Curse of Dimensionality: The *curse of dimensionality* poses a significant challenge for k-NN in high-dimensional spaces. As the number of dimensions (d) increases:

- The relative distances between points become less meaningful
- The sparsity of data grows, requiring exponentially more training samples



To mitigate the curse of dimensionality, dimensionality reduction techniques such as PCA or feature selection methods can be applied before using k-NN.

Computational Complexity: A key drawback of k-NN is its computational cost. For each query, the distance to all N training points must be computed, making it inefficient for large datasets. Optimization techniques such as KD-trees or approximate nearest neighbor search can alleviate this issue.

Visualization: The diagram below illustrates a query point, its nearest neighbors, and the centroids of two classes. This example demonstrates the decision-making process in k-NN.

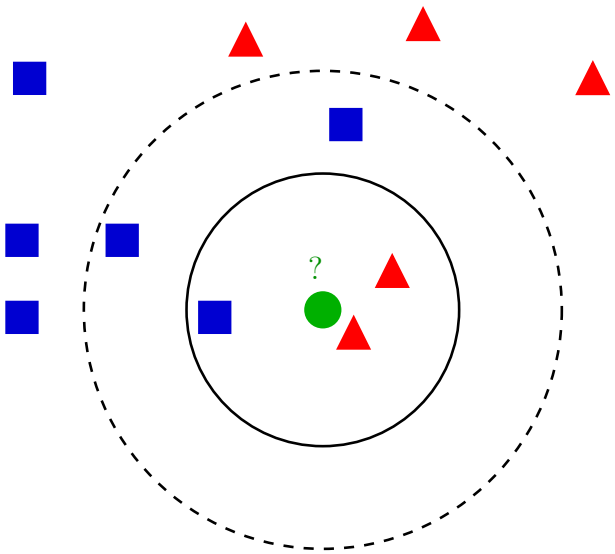


Figure 5: The test sample (green dot) should be classified either to blue squares or to red triangles. If $k = 3$ (solid line circle) it is assigned to the red triangles because there are 2 triangles and only 1 square inside the inner circle.

2.9 Ensemble Learning

Ensemble learning refers to a machine learning paradigm where multiple models (referred to as "learners") are combined to solve a particular problem. The goal of ensemble methods is to improve the predictive performance by leveraging the strengths of individual learners and mitigating their weaknesses. Two popular ensemble techniques are bagging and boosting.

2.9.1 Bagging

Bagging, or Bootstrap Aggregating, is an ensemble method that builds multiple versions of a model by training each one on a different **bootstrapped** subset of the training data. These subsets are created by randomly sampling the dataset with replacement. Once trained, the predictions of the individual models are aggregated to produce the final output. For regression problems, the aggregation is typically a simple average of predictions, whereas for classification problems, majority voting is commonly used.
Mathematically, let $\{D_1, D_2, \ldots, D_k\}$ be the bootstrapped datasets generated from the original training dataset D. For each i, a model M_i is trained on D_i. The final prediction $\hat{y}$ for an input x is given by:

$$\hat{y} = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} M_i(x) & \text{for regression,} \\ \text{mode}(\{M_1(x), M_2(x), \ldots, M_k(x)\}) & \text{for classification.} \end{cases}$$

Bagging reduces variance by averaging predictions over multiple learners, thereby improving stability and accuracy, especially in models prone to overfitting (e.g., decision trees).

2.9.2 Boosting

Boosting is an iterative ensemble technique that sequentially trains models, with each model focusing on the **hardest** examples missed by its predecessors. In this process,

weak learners are combined to create a strong learner. Unlike bagging, **boosting assigns weights** to training examples to prioritize correcting errors made by previous models.

Hardest Examples in Boosting The "hardest examples" in boosting are the data points that previous learners consistently misclassify or predict with high error. Boosting emphasizes these examples by increasing their weights in the training process, ensuring that subsequent models pay more attention to them.

Weighted Mean in Boosting In boosting, the final prediction is a weighted mean of the outputs of individual learners. Let M_i be the i-th weak learner with weight α_i (reflecting its performance), and $M_i(x)$ be the prediction of M_i for input x. The weighted mean for the final prediction $\hat{y}$ is given by:

$$\hat{y} = \text{sign}\left(\sum_{i=1}^{k} \alpha_i M_i(x)\right),$$

where the sign function is defined as:

$$\text{sign}(z) = \begin{cases} +1 & \text{if } z > 0, \\ 0 & \text{if } z = 0, \\ -1 & \text{if } z < 0. \end{cases}$$

Defining Error Mathematically The error of a weak learner M_i is defined as the weighted sum of the misclassified examples. Let w_j be the weight of example j in the training data, and let $I(M_i(x_j) \neq y_j)$ be an indicator function that equals 1 if M_i misclassifies x_j, and 0 otherwise. The error ϵ_i is given by:

$$\epsilon_i = \sum_{j=1}^{n} w_j \cdot I(M_i(x_j) \neq y_j).$$

Weak Learners A weak learner is defined as a learning algorithm that performs slightly better than random guessing, regardless of the data distribution. Mathematically, this means that for a binary classification task, the weak learner achieves an error rate $\epsilon < 0.5$, where ϵ is slightly less than 0.5 (i.e., better than random guessing).

This implies that for any distribution of examples, the weak learner is able to produce a hypothesis $M(x)$ such that:

$$P(M(x) = y) > 0.5,$$

where $P(M(x) = y)$ represents the probability that the weak learner correctly predicts the label y.

Role of Small ϵ In boosting, ϵ denotes the error of a weak learner, and $\epsilon < 0.5$ ensures that the learner is better than random guessing. A small ϵ means the learner has a low error rate, which allows boosting to focus on the hardest examples while maintaining steady progress in reducing overall error. As long as the weak learner consistently achieves $\epsilon < 0.5$, the boosting algorithm can converge to a strong learner.

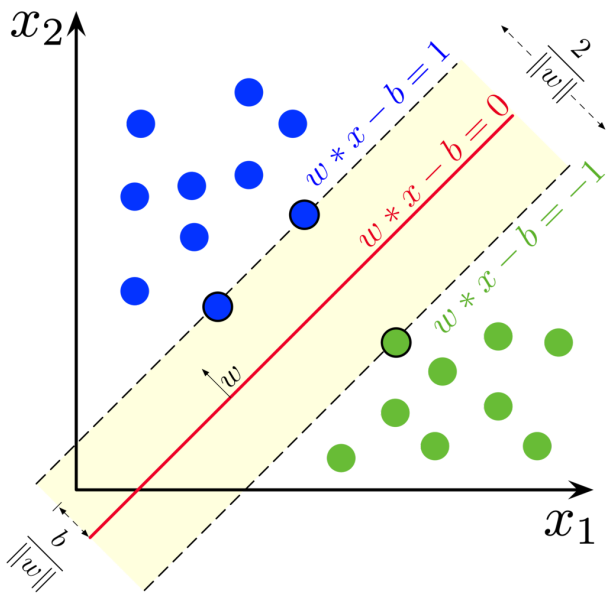


Figure 6: SVM Visualization: Hyperplane, margins, and support vectors.

2.10 Support Vector Machines

Definition:

SVMs are supervised learning models used for classification and regression tasks. The main goal of SVMs is to **find a hyperplane that best separates data points from different classes with the maximum margin.** The margin is defined as the distance between the hyperplane and the closest data points, known as support vectors.

SVMs can handle both linearly separable and non-linearly separable data. For non-linearly separable data, kernel functions, such as the polynomial kernel and radial basis function (RBF), are used to map the data into higher-dimensional spaces where it becomes linearly separable.

Mathematical Definition:

Given a training dataset $\{(\mathbf{x}_i, y_i) \mid i = 1, 2, \ldots, N\}$ where $\mathbf{x}_i \in \mathbb{R}^d$ are input feature vectors and $y_i \in \{-1, 1\}$ are class labels, the SVM solves the following optimization problem:

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

Subject to:

$$y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 \quad \forall i.$$

The hyperplane is defined as:

$$\mathbf{w}^\top \mathbf{x} + b = 0.$$

To solve the optimization problem, we often use the dual form, which introduces Lagrange multipliers α_i for each data point. The dual formulation of SVM is:

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j)$$

Subject to:

$$\sum_{i=1}^{N} \alpha_i y_i = 0, \quad \alpha_i \geq 0 \quad \forall i,$$

where $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ is the **kernel** function, which computes the similarity between data points. Once the optimization is solved, the decision boundary is computed as:

$$f(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i y_i \kappa(\mathbf{x}_i, \mathbf{x}) + b,$$

where only the support vectors (points with $\alpha_i > 0$) contribute to the sum. The margin is maximized by minimizing $\|\mathbf{w}\|^2$, ensuring robust separation between the two classes.

SVMs can also be extended to handle non-linear boundaries using kernel functions $\kappa(\mathbf{x}_i, \mathbf{x}_j)$, which implicitly map the data into higher-dimensional feature spaces.

2.11 Kernel Functions

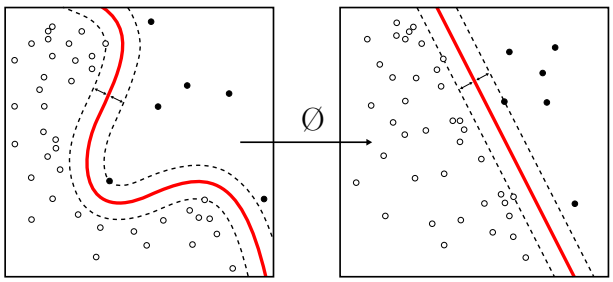


Figure 7: Kernel machine Visualization

Kernel functions are a fundamental concept in Support Vector Machines (SVMs) and other machine learning algorithms. They enable SVMs to solve problems where data is not linearly separable by implicitly mapping the input features into a higher-dimensional space without explicitly computing the transformation. This process is known as the **kernel trick**.

Definition: Given two input vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, a kernel function $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ computes the dot product of their mapped representations in a higher-dimensional feature space Φ:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)^\top \Phi(\mathbf{x}_j).$$

By using a kernel function, SVMs avoid the computational expense of explicitly mapping data into a higher-dimensional space, allowing them to efficiently handle complex, non-linear decision boundaries.

Common Kernel Functions:

1. **Linear Kernel** The simplest kernel, used for linearly separable data:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^\top \mathbf{x}_j.$$

   This kernel does not transform the data and directly uses the original feature space.

2. **Polynomial Kernel** Captures polynomial relationships between features:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^\top \mathbf{x}_j + c)^d,$$

   where $c \geq 0$ is a constant and d is the degree of the polynomial.

3. **Radial Basis Function (RBF) Kernel** A popular kernel for non-linear data, based on the Gaussian function:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2\right),$$

   where $\gamma > 0$ controls the width of the kernel. This kernel maps data into an infinite-dimensional space.

4. **Sigmoid Kernel** Inspired by neural networks, this kernel is defined as:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\beta \mathbf{x}_i^\top \mathbf{x}_j + c),$$

   where $\beta > 0$ and c are kernel parameters. It is less commonly used than RBF or polynomial kernels.

Choosing a Kernel Function: The choice of kernel function depends on the nature of the data and the problem at hand:

- Use the **linear kernel** if the data is linearly separable or the number of features is very large.
- Use the **RBF kernel** as a default choice for most non-linear problems.
- Use the **polynomial kernel** for capturing specific polynomial relationships between features.
- Experiment with hyperparameters such as γ, c, and d to optimize the performance of the SVM.

Back to Boosting: Boosting and SVMs can complement each other in several ways:

- **Weak SVMs in Boosting:** Boosting can use simpler SVMs (e.g., with fewer support vectors or linear kernels) as weak learners, refining the decision boundary iteratively by focusing on misclassified examples.
- **Hardest Examples:** Boosting emphasizes the hardest examples by increasing their weights, aligning with SVMs' focus on examples near or violating the margin.
- **Kernel Combination:** Boosting can combine multiple kernel functions in SVMs, optimizing their weights to learn a better decision boundary (as in Multiple Kernel Learning).
- **Overfitting Reduction:** By combining predictions across learners, boosting reduces overfitting when using SVMs as base models.

2.12 Computational Learning Theory

Definitions:

- **Computational complexity**: The resources (time, space, etc.) required to learn a target concept.
- **Sample complexity**: The number of examples needed to learn the target concept to a given level of accuracy and confidence.
- **Mistake bounds**: The maximum number of mistakes a learning algorithm can make during the learning process.

2.12.1 PAC (Probably Approximately Correct) Learning:

- **Training error**: The fraction of training examples misclassified by hypothesis h.
- **True error**: The fraction of examples misclassified by h on a sample drawn from the underlying distribution D.
- $error_D(h) = \Pr_{x \sim D}[c(x) \neq h(x)]$, where $c(x)$ is the target concept.

Notation:

- C: Concept class (set of all target concepts).
- L: Learner (learning algorithm).
- H: Hypothesis space (set of possible hypotheses).
- $n = |H|$: Size of the hypothesis space.
- D: Distribution over inputs.
- ϵ: Error goal, where $0 \leq \epsilon \leq \frac{1}{2}$.
- δ: Certainty goal, where $0 \leq \delta \leq \frac{1}{2}$.

Learnability: The concept class C is PAC-learnable by learner L using hypothesis space H if, with probability at least $1-\delta$, the learner L outputs a hypothesis $h \in H$ such that $error_D(h) \leq \epsilon$. This must be achievable in time and sample size polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and n. **In other words, PAC learnability means the learning algorithm can produce a hypothesis that is both accurate and reliable, without requiring an unreasonable amount of data or computational resources**.

2.12.2 ε-exhausted Version Space

Definition: The version space is the set of all hypotheses in the hypothesis space H that are consistent with the training data. Formally, the version space is defined as:

$$VS = \{h \in H \mid h(x_i) = c(x_i) \text{ for all training examples } (x_i, c(x_i))\}.$$

An **ε-exhausted version space** means that the remaining hypotheses in the version space are all approximately correct, such that the true error of any hypothesis $h \in VS$ is at most ε.

Properties:

- If the version space is ε-exhausted, any hypothesis h selected from the version space will have an error bounded by ε.
- Achieving an ε-exhausted version space typically requires a sufficient number of training examples to rule out hypotheses with error greater than ε.
- This concept ensures the learner can make confident predictions within the specified error tolerance.

Connection to PAC Learning: In PAC learning, the goal is to efficiently find a hypothesis h such that $error_D(h) \leq \varepsilon$ with high probability $(1 - \delta)$. An ε-exhausted version space guarantees that any consistent hypothesis meets this criterion, provided sufficient training data are available.

2.12.3 Haussler's Theorem

Haussler's Theorem provides a bound on the sample complexity required for PAC learning in terms of the size of the hypothesis space H. Specifically, it states that:
If the hypothesis space H has finite cardinality $|H| = n$, and we aim to achieve a true

error of at most ε with confidence at least $1 - \delta$, then the number of training examples m required satisfies:

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

Interpretation:

- The sample complexity grows with the size of the hypothesis space $|H|$, the desired accuracy ε, and the confidence level $1 - \delta$.
- Larger hypothesis spaces (more complex models) require more training examples to ensure PAC learnability.
- For fixed ε and δ, reducing the hypothesis space H (e.g., through regularization or constraints) reduces the required sample size.

Key Implications:

- Haussler's Theorem ensures that PAC learning is feasible for finite hypothesis spaces, provided the number of training examples scales logarithmically with $|H|$ and $\frac{1}{\delta}$.
- The theorem highlights the trade-off between the complexity of the hypothesis space and the amount of training data needed to achieve reliable learning.

Connection to VC Dimension: Haussler's Theorem is related to bounds on sample complexity using the VC dimension for infinite hypothesis spaces, where the growth of the hypothesis space is characterized by its capacity to shatter data points rather than its cardinality.

2.12.4 VC Dimension

Definition: The **VC (Vapnik–Chervonenkis) dimension** of a hypothesis space H is the maximum number of points that can be *shattered* by hypotheses in H. A set of points is said to be shattered if, for every possible labeling of the points, there exists a hypothesis in H consistent with that labeling.

Formally:

$$\mathrm{VC}(H) = \max\{m \mid \exists \text{ a set of } m \text{ points shattered by } H\}.$$

If no finite m exists such that a set of m points can be shattered, the VC dimension is infinite.

Examples:

- **Linear separators in 2D:** A linear separator (e.g., a line) in a 2D plane has a VC dimension of 3. It can shatter any set of 3 points in general position (not collinear), but it cannot shatter 4 points arranged in a general configuration.
- **The ring:** A ring-shaped hypothesis space (e.g., deciding whether a point lies inside or outside a circular boundary) has a VC dimension of 3. It can shatter 3 points (e.g., inside, outside, inside) but not 4.
- **Polygons:** The VC dimension of a hypothesis space that uses polygons depends on the number of sides of the polygon. For example, if a hypothesis space includes all triangles, its VC dimension is 7, as it can shatter 7 points in a plane. Increasing the number of sides increases the VC dimension.

Sample Complexity: The VC dimension determines the number of training examples m required to PAC-learn a concept class with confidence $1 - \delta$ and error goal ε. For a hypothesis space H with VC dimension d:

$$m \geq \frac{d}{\varepsilon} \ln \frac{1}{\varepsilon} + \frac{1}{\varepsilon} \ln \frac{1}{\delta}.$$

Case of Finite Hypothesis Space: When H is finite with size $|H| = n$, the VC dimension d satisfies $d \leq \log_2(n)$. In this case, Haussler's Theorem provides a bound on the sample complexity:

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

Importance of VC Dimension:

- The VC dimension provides a measure of the expressiveness or capacity of the hypothesis space.
- Larger VC dimensions allow the hypothesis space to fit more complex patterns but require more training data to avoid overfitting.
- For infinite hypothesis spaces (e.g., linear separators in higher dimensions), the VC dimension ensures sample complexity remains bounded.

2.13 Bayes Rule

Bayes Rule is a fundamental formula in probability that allows us to update our beliefs about an event based on new evidence. It is given by:

$$P(a \mid b) = \frac{P(b \mid a) \cdot P(a)}{P(b)}$$

Where:

- $P(a \mid b)$: Posterior probability of a given b (updated belief after observing evidence b).
- $P(b \mid a)$: Likelihood of evidence b occurring if a is true.
- $P(a)$: Prior probability of a (belief about a before observing evidence).
- $P(b)$: Marginal probability of b (total probability of evidence b across all possible causes).

2.13.1 Example: Alien Detection

Imagine you are the captain of the spaceship *Intergalactic Voyager*, exploring deep space. Your ship's sensors are designed to detect alien ships in nearby regions of the galaxy. However, the sensors are imperfect.

- A: Event that there is an alien ship nearby.
- S: Event that the sensor detects alien signals.

Your mission is to calculate the probability of aliens being nearby $P(A \mid S)$ given that the sensors have detected a signal.

Known Information:

- $P(A) = 0.02$: The prior probability of an alien ship being nearby (aliens are rare!).
- $P(S \mid A) = 0.9$: The likelihood that the sensor detects signals when aliens are present.
- $P(S \mid A^c) = 0.1$: The likelihood that the sensor detects signals even when there are no aliens (false positive rate).
- $P(A^c) = 0.98$: The prior probability of no aliens nearby.

Step 1: Total Probability of Sensor Detection

We first calculate the total probability of the sensor detecting a signal, $P(S)$:

$$P(S) = P(S \mid A) \cdot P(A) + P(S \mid A^c) \cdot P(A^c)$$

Substitute the values:

$$P(S) = (0.9 \cdot 0.02) + (0.1 \cdot 0.98) = 0.018 + 0.098 = 0.116$$

Step 2: Posterior Probability of Aliens Nearby

Now, we use Bayes' Rule to calculate $P(A \mid S)$:

$$P(A \mid S) = \frac{P(S \mid A) \cdot P(A)}{P(S)}$$

Substitute the values:

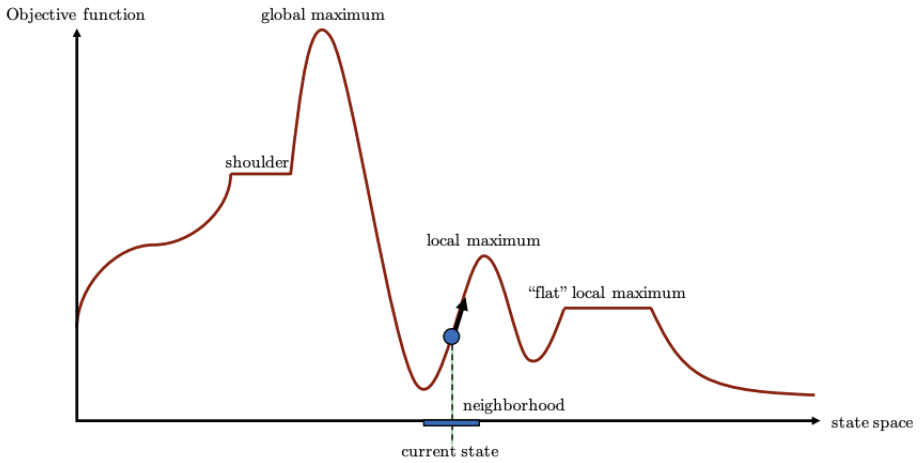
$$P(A \mid S) = \frac{0.9 \cdot 0.02}{0.116} \approx 0.155$$

Even with a positive signal, there is only a 15.5% chance that an alien ship is nearby due to the rarity of alien encounters.

2.14 Bayesian Learning

3 Unsupervised Learning

3.1 Hill Climbing



Hill Climbing is an optimization algorithm used to solve problems by iteratively improving a candidate solution based on its neighboring solutions. It is a type of greedy algorithm that aims to find the global maximum (or minimum) of an objective function, but it can sometimes get stuck in local optima.

3.1.1 Algorithm Overview

The Hill Climbing algorithm works as follows:

1. **Initialization:** Start with an initial solution, often chosen randomly.
2. **Evaluation:** Compute the value of the objective function for the current solution.
3. **Move to Neighbor:** Explore the neighboring solutions of the current state and move to the neighbor with the best objective value.
4. **Repeat:** Repeat the process until a stopping condition is met, such as a maximum number of iterations or no improvement in the objective function.

3.1.2 Key Characteristics

- **Deterministic:** Hill Climbing is deterministic, meaning it follows a specific path based on the objective function and current state.
- **Local Search:** The algorithm searches only the immediate neighborhood of the current solution.
- **Risk of Local Optima:** Hill Climbing may get stuck in a local maximum or minimum if it cannot escape to explore other regions of the solution space.

3.1.3 Variants of Hill Climbing

Several variants of the basic Hill Climbing algorithm exist to improve its performance and reduce its limitations:

- **Stochastic Hill Climbing:** Instead of always choosing the best neighbor, this variant selects a random neighbor to explore, introducing randomness to escape local optima.
- **First-Choice Hill Climbing:** Evaluates neighbors one by one in a random order and moves to the first neighbor that improves the objective function.
- **Simulated Annealing:** Adds a probability of accepting worse solutions at the beginning, which decreases over time, allowing the algorithm to escape local optima.

3.2 Randomized Hill Climbing

Randomized Hill Climbing is a variant of the standard Hill Climbing algorithm that incorporates **randomness** to improve its ability to **escape local optima** and explore the solution space more effectively. Unlike standard Hill Climbing, which always selects the best neighboring solution, Randomized Hill Climbing introduces randomness in selecting and evaluating neighbors.

3.2.1 Algorithm Overview

The Randomized Hill Climbing algorithm works as follows:

1. **Initialization:** Start with an initial solution, often chosen randomly.
2. **Evaluation:** Compute the value of the objective function for the current solution.
3. **Random Neighbor Selection:** Randomly select a neighbor from the solution space instead of evaluating all neighbors.
4. **Acceptance Criterion:** Move to the selected neighbor if it improves the objective function. Optionally, accept worse neighbors with a small probability to escape local optima.
5. **Repeat:** Repeat the process until a stopping condition is met, such as a maximum number of iterations or no improvement in the objective function.

3.2.2 Key Characteristics

- **Exploration:** Randomized Hill Climbing increases exploration by considering random neighbors instead of deterministic moves.
- **Escape Local Optima:** Randomness helps the algorithm avoid getting stuck in local optima by occasionally accepting suboptimal solutions.
- **Flexibility:** It is less likely to converge prematurely compared to standard Hill Climbing, making it suitable for complex optimization landscapes.

3.2.3 Advantages and Disadvantages

Advantages:

- Can escape local optima more effectively than standard Hill Climbing.
- Provides a balance between exploration and exploitation.
- Simple to implement and extend.

Disadvantages:

- Random moves may lead to slower convergence compared to deterministic Hill Climbing.
- Highly dependent on the choice of random neighbor selection and acceptance criteria.

Randomized Hill Climbing is particularly effective for problems with rugged or complex optimization landscapes where local optima are common.

3.3 Simulated Annealing

Simulated Annealing (SA) is an optimization algorithm inspired by the physical process of annealing in metallurgy. In annealing, a material is heated to a high temperature and then cooled slowly to allow its particles to settle into a low-energy, highly ordered state. Simulated Annealing mimics this process to **find a near-optimal solution for optimization problems, particularly those with complex and rugged solution spaces**.

3.3.1 Key Idea

The key idea of Simulated Annealing is to balance **exploration** and **exploitation** by occasionally allowing moves to worse solutions with a probability that decreases as the "temperature" of the system is reduced. This enables the algorithm to escape local optima and explore a broader region of the solution space.

3.3.2 Algorithm Overview

The Simulated Annealing algorithm proceeds as follows:

1. **Initialization:** Start with an initial solution x_0, an initial temperature T_0, and a cooling schedule.
2. **Iterative Optimization:**
   (a) Evaluate the objective function $f(x)$ at the current solution x.
   (b) Generate a new solution x' in the neighborhood of x.
   (c) Compute the change in the objective function: $\Delta f = f(x') - f(x)$.
   (d) If $\Delta f > 0$, accept x' as the new solution.
   (e) If $\Delta f \leq 0$, accept x' with probability:

$$P = \exp\left(\frac{\Delta f}{T}\right),$$

   where T is the current temperature.
3. **Cooling:** Reduce the temperature T according to the cooling schedule.
4. **Stopping Criterion:** Stop when T reaches a predefined minimum value or after a maximum number of iterations.

3.3.3 Cooling Schedule

The cooling schedule determines how the temperature decreases over time. Common schedules include:

- **Exponential Cooling:** $T = T_0 \cdot \alpha^k$, where $0 < \alpha < 1$ is the cooling rate and k is the iteration number.
- **Linear Cooling:** $T = T_0 - \beta \cdot k$, where β is the cooling decrement.
- **Logarithmic Cooling:** $T = \frac{T_0}{\log(k+1)}$, where the temperature decreases slowly as the iterations progress.

3.3.4 Advantages and Disadvantages

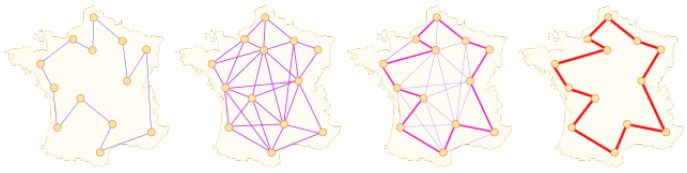
Advantages:

- Can escape local optima, making it effective for multimodal optimization problems.
- Flexible and easy to adapt to various types of optimization problems.
- Does not require gradient information, so it can handle non-differentiable or noisy objective functions.

Disadvantages:

- Requires careful tuning of parameters like the initial temperature, cooling schedule, and stopping criteria.
- Slower convergence compared to greedy algorithms for simple problems.
- No guarantee of finding the global optimum, especially with a poorly designed cooling schedule.

3.4 Genetic Algorithms



Genetic Algorithms (GAs) are a class of optimization algorithms inspired by the process of natural selection in biological evolution. They are particularly well-suited for solving complex problems where the search space is large, multimodal, or poorly understood. GAs simulate evolution by iteratively selecting and improving candidate solutions through genetic operations like selection, crossover, and mutation.

3.4.1 Key Idea

The central idea of Genetic Algorithms is to represent candidate solutions as "chromosomes" and iteratively improve them by mimicking the biological processes of evolution. The algorithm maintains a population of solutions, selects the fittest individuals, combines them to produce offspring, and introduces random mutations to maintain diversity.

3.4.2 Algorithm Overview

The Genetic Algorithm proceeds as follows:

1. **Initialization:** Generate an initial population of N individuals (solutions), typically chosen randomly.
2. **Evaluation:** Compute the fitness of each individual using a fitness function that evaluates the quality of the solution.
3. **Selection:** Select individuals for reproduction based on their fitness. Higher-fitness individuals are more likely to be selected. Common selection methods include:
    - **Roulette Wheel Selection:** Probability of selection is proportional to fitness.
    - **Tournament Selection:** Randomly select a subset of individuals and choose the best among them.

- **Rank Selection:** Rank individuals by fitness and assign selection probabilities based on rank.

4. **Crossover (Recombination):** Combine pairs of parent solutions to create offspring by exchanging genetic information. Common crossover techniques include:
   - **Single-Point Crossover:** Split the parent chromosomes at a single point and exchange the segments.
   - **Two-Point Crossover:** Split the chromosomes at two points and exchange the middle segments.
   - **Uniform Crossover:** Exchange genes randomly between parents.
5. **Mutation:** Introduce random changes to the offspring to maintain diversity and avoid premature convergence. Mutation typically flips bits or alters genes with a small probability.
6. **Replacement:** Replace the current population with the offspring (and sometimes a subset of the best individuals from the current population, known as elitism).
7. **Repeat:** Iterate through the evaluation, selection, crossover, mutation, and replacement steps until a stopping condition is met, such as a maximum number of generations or convergence to an optimal solution.

3.4.3 Key Components

- **Population:** A set of candidate solutions.
- **Fitness Function:** Evaluates the quality of a solution and guides the evolution process.
- **Genetic Operators:** Selection, crossover, and mutation drive the evolution of the population.
- **Stopping Criteria:** Maximum number of generations, fitness threshold, or lack of significant improvement.

3.4.4 Advantages and Disadvantages

Advantages:

- Highly flexible and can handle complex, multimodal, and non-differentiable search spaces.
- Robust to noisy fitness landscapes and avoids getting stuck in local optima.
- Can optimize multiple objectives simultaneously (multi-objective optimization).

Disadvantages:

- Computationally expensive due to the evaluation of many individuals in each generation.
- Requires careful tuning of parameters like population size, crossover rate, and mutation rate.
- May converge slowly, especially for problems with large solution spaces.

3.4.5 Example: GA for Solving the Traveling Salesperson Problem

The Traveling Salesperson Problem (TSP) seeks the shortest route that visits each city exactly once and returns to the starting city. A Genetic Algorithm can solve the TSP as follows:

1. **Representation:** Represent each solution (chromosome) as a sequence of city visits.
2. **Fitness Function:** Evaluate the total distance of the tour. Lower distances correspond to higher fitness.
3. **Crossover:** Use ordered crossover to ensure valid tours (e.g., preserve city sequences while exchanging segments).
4. **Mutation:** Swap two cities randomly or reverse the order of a segment of the tour.
5. **Selection:** Use tournament selection to prioritize shorter tours.
6. **Iterate:** Evolve the population over several generations until convergence.

3.5 MIMIC

MIMIC, or *Mutual Information Maximization for Input Clustering*, is a probabilistic algorithm used in machine learning for optimization and modeling dependency structures among variables. It belongs to the family of Estimation of Distribution Algorithms (EDAs), which replace traditional genetic algorithm operators (e.g., crossover and mutation) with the estimation and sampling of a probability distribution.

The core idea of MIMIC is to model the dependencies between variables by maximizing the mutual information between a subset of selected variables and the remaining variables. In MIMIC, the algorithm iteratively constructs a chain-like dependency model (a greedy approximation of a Bayesian network) by selecting variables that maximize the mutual information with the previously selected variables. This process can be summarized as follows:

1. Initialize with a population of candidate solutions.
2. Compute the entropy and mutual information for all variables based on the best-performing solutions.
3. Select the variable with the lowest entropy as the starting point.

4. Iteratively add variables that maximize mutual information with the already selected variables, forming a permutation.

5. Build a probabilistic model (e.g., a chain) based on this permutation.

6. Sample new solutions from this model and evaluate them.

7. Repeat until convergence or a stopping criterion is met.

MIMIC is particularly effective for problems with strong inter-variable dependencies, offering a balance between computational efficiency and modeling accuracy. However, its greedy approach may lead to suboptimal solutions in complex, highly multimodal optimization landscapes.

3.6 Information Theory

Information theory provides a mathematical framework for quantifying information, uncertainty, and dependencies between variables. Below, we explore key concepts related to the information between two variables, including joint entropy, conditional entropy, mutual information, and the Kullback-Leibler divergence.

3.6.1 Joint Entropy and Conditional Entropy

For two random variables X and Y, the *joint entropy* $H(X, Y)$ measures the total uncertainty associated with their joint distribution. It is defined as:

$$H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} P(x, y) \log P(x, y),$$

where $P(x, y)$ is the joint probability mass function of X and Y.

The *conditional entropy* $H(X|Y)$ quantifies the uncertainty in X given knowledge of Y, calculated as:

$$H(X|Y) = -\sum_{x \in X} \sum_{y \in Y} P(x, y) \log P(x|y),$$

where $P(x|y)$ is the conditional probability of X given Y. It follows that:

$$H(X, Y) = H(Y) + H(X|Y).$$

3.6.2 Mutual Information

Mutual information $I(X; Y)$ measures the amount of information shared between X and Y, or the reduction in uncertainty about one variable given knowledge of the other. It is defined as:

$$I(X; Y) = H(X) - H(X|Y),$$

or equivalently:

$$I(X; Y) = \sum_{x \in X} \sum_{y \in Y} P(x, y) \log \frac{P(x, y)}{P(x)P(y)}.$$

Mutual information is symmetric ($I(X;Y) = I(Y;X)$) and non-negative, with $I(X;Y) = 0$ if and only if X and Y are independent.

3.6.3 Two Independent Coins

Consider two fair coins, X and Y, each with outcomes {H, T} and probabilities $P(X = H) = P(X = T) = 0.5$, and similarly for Y. If the coins are independent, the joint probability is:

$$P(X, Y) = P(X)P(Y),$$

e.g., $P(X = H, Y = T) = 0.5 \times 0.5 = 0.25$. The joint entropy is:

$$H(X, Y) = -4 \times (0.25 \log 0.25) = 2 \,\text{bits},$$

since there are four equally likely outcomes. The individual entropies are $H(X) = H(Y) = 1 \,\text{bit}$, and the conditional entropy $H(X|Y) = H(X) = 1 \,\text{bit}$ because independence implies no information gain. Thus:

$$I(X;Y) = H(X) - H(X|Y) = 1 - 1 = 0,$$

confirming that mutual information is zero for independent variables.

3.6.4 Kullback-Leibler Divergence

The *Kullback-Leibler (KL) divergence* $D_{KL}(P||Q)$ measures the difference between two probability distributions P and Q over the same variable. It is defined as:

$$D_{KL}(P||Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)},$$

where P is the true distribution and Q is an approximation. KL divergence is non-negative ($D_{KL}(P||Q) \geq 0$), with equality if and only if $P = Q$, but it is not symmetric ($D_{KL}(P||Q) \neq D_{KL}(Q||P)$). It is often interpreted as the "information loss" when using Q to approximate P, and it connects to mutual information via:

$$I(X;Y) = D_{KL}(P(X, Y)||P(X)P(Y)).$$

These concepts form the foundation of information theory, with applications in machine learning, data compression, and statistical inference.

3.7 Clustering & EM

3.7.1 Single Linkage Clustering

Single linkage clustering, also known as *nearest neighbor clustering*, is a method of agglomerative hierarchical clustering. It starts with each data point as its own cluster and iteratively merges the two clusters with the smallest minimum pairwise distance until all points belong to a single cluster. The distance between two clusters C_i and C_j is defined as:

$$d(C_i, C_j) = \min_{x \in C_i, y \in C_j} \|x - y\|,$$

where $\|x - y\|$ is a distance metric (e.g., Euclidean distance) between points x and y in clusters C_i and C_j, respectively.

The algorithm proceeds as follows:

1. Initialize n clusters, one for each of the n data points.
2. Compute the pairwise distances between all clusters.
3. Merge the two clusters with the smallest $d(C_i, C_j)$.
4. Update the distance matrix by recalculating distances between the new cluster and all remaining clusters using the single linkage criterion.
5. Repeat steps 3 and 4 until only one cluster remains.

The result is a dendrogram, a tree-like structure that visualizes the hierarchy of merges and allows the user to choose a desired number of clusters by cutting the tree at a specific height. Single linkage clustering is computationally efficient with a time complexity of $O(n^2)$ when implemented with a priority queue. However, it is sensitive to noise and outliers, often producing elongated, "chain-like" clusters due to its focus on minimum distances. This chaining effect can be a drawback in datasets with irregular cluster shapes.

3.7.2 K-means Clustering

K-means clustering is a popular partitional clustering algorithm that aims to divide a set of n data points into K distinct, non-overlapping clusters, where each point belongs to the cluster with the nearest mean (centroid). The algorithm minimizes the within-cluster variance, often measured as the sum of squared distances between points and their assigned cluster centroids.

Given a dataset $X = \{x_1, x_2, \ldots, x_n\}$, where each $x_i \in \mathbb{R}^d$, and a predefined number of clusters K, the objective is to find a partitioning $C = \{C_1, C_2, \ldots, C_K\}$ and centroids $\mu = \{\mu_1, \mu_2, \ldots, \mu_K\}$ that minimize the cost function:

$$J = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2,$$

where $\|x_i - \mu_k\|$ is the Euclidean distance between a data point x_i and the centroid μ_k of cluster C_k.

The K-means algorithm operates iteratively as follows:

1. **Initialization**: Randomly select K initial centroids (e.g., by choosing K points from the dataset or using a method like K-means++ for better initialization).
2. **Assignment Step**: Assign each data point x_i to the cluster C_k with the nearest centroid:
$$C_k = \{x_i : \|x_i - \mu_k\| \leq \|x_i - \mu_j\| \text{ for all } j \neq k\}.$$

3. **Update Step**: Recalculate the centroid of each cluster as the mean of all points assigned to it:
$$\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i,$$
where $|C_k|$ is the number of points in cluster C_k.

4. **Convergence Check**: Repeat steps 2 and 3 until the centroids no longer change significantly or a maximum number of iterations is reached.

The algorithm converges to a local optimum, and the final result depends on the initial centroid placement, making it sensitive to initialization. To mitigate this, multiple runs with different initializations are often performed, selecting the solution with the lowest J.
K-means has a time complexity of $O(nKdI)$, where I is the number of iterations, and is efficient for large datasets. However, it assumes spherical, equally sized clusters and requires K to be specified in advance, which can be a limitation. Techniques like the elbow method, analyzing the plot of J versus K, are commonly used to determine an optimal K.

3.7.3 Soft Clustering

Soft clustering, also known as *fuzzy clustering*, is an approach to clustering where data points are not strictly assigned to a single cluster (as in hard clustering methods like K-means), but instead have varying degrees of membership across multiple clusters. This probabilistic or fuzzy assignment reflects the uncertainty or overlap often present in real-world data, allowing for more nuanced interpretations of cluster boundaries.

In soft clustering, each data point x_i in a dataset $X = \{x_1, x_2, \ldots, x_n\}$ is associated with a membership vector $u_i = [u_{i1}, u_{i2}, \ldots, u_{iK}]$, where $u_{ik} \in [0, 1]$ represents the degree of membership of x_i in cluster k, and K is the number of clusters. Typically, the memberships satisfy the constraint:
$$\sum_{k=1}^{K} u_{ik} = 1,$$
ensuring that the total membership of a point across all clusters sums to 1.

A prominent example of soft clustering is the *Fuzzy C-Means (FCM)* algorithm, which extends K-means by introducing fuzziness. FCM minimizes the following objective function:
$$J_m = \sum_{i=1}^{n} \sum_{k=1}^{K} u_{ik}^m \|x_i - \mu_k\|^2,$$

where μ_k is the centroid of cluster k, $\|x_i - \mu_k\|$ is the Euclidean distance, and $m > 1$ is a fuzziness parameter controlling the softness of assignments (typically $m = 2$). The higher the m, the fuzzier the boundaries between clusters.

The FCM algorithm iterates through two main steps:

1. **Membership Update**: Compute the membership u_{ik} for each point x_i and cluster k as:
$$u_{ik} = \frac{1}{\sum_{j=1}^{K} \left(\frac{\|x_i - \mu_k\|}{\|x_i - \mu_j\|} \right)^{\frac{2}{m-1}}},$$
reflecting the inverse distance to each centroid, weighted by the fuzziness parameter.

2. **Centroid Update**: Recalculate each cluster centroid μ_k as a weighted average:
$$\mu_k = \frac{\sum_{i=1}^{n} u_{ik}^m x_i}{\sum_{i=1}^{n} u_{ik}^m},$$
where the weights are the memberships raised to the power m.

3. **Convergence**: Repeat until the memberships or centroids stabilize, or a maximum number of iterations is reached.

Soft clustering is closely related to probabilistic models like the Expectation-Maximization (EM) algorithm applied to Gaussian Mixture Models (GMMs), where memberships are interpreted as posterior probabilities $P(k|x_i)$. Unlike FCM, GMMs assume an underlying generative model (e.g., Gaussian distributions) and optimize likelihood rather than a heuristic cost function.

Soft clustering excels in applications with overlapping clusters, such as image segmentation, gene expression analysis, or customer segmentation, where data points may naturally belong to multiple groups. However, it is computationally more intensive than hard clustering (e.g., K-means) and requires choosing parameters like K and m, which can impact results.

3.7.4 Maximum Likelihood Gaussian

Maximum Likelihood Estimation (MLE) is a statistical method used to estimate the parameters of a probability distribution by maximizing the likelihood of observing the given data. For a Gaussian (normal) distribution, MLE is commonly applied to determine the mean and variance that best describe a dataset, a concept foundational to clustering techniques like Gaussian Mixture Models (GMMs) used in soft clustering.

Suppose we have a dataset $X = \{x_1, x_2, \ldots, x_n\}$, where each $x_i \in \mathbb{R}$ is assumed to be independently drawn from a univariate Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$. The probability density function for a single observation x_i is:
$$p(x_i|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_i - \mu)^2}{2\sigma^2} \right).$$
The likelihood function for the entire dataset, assuming independence, is the product of the individual densities:
$$L(\mu, \sigma^2) = \prod_{i=1}^{n} p(x_i|\mu, \sigma^2) = \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right)^n \exp\left(-\frac{1}{2\sigma^2} \sum_{i=1}^{n} (x_i - \mu)^2 \right).$$
To simplify optimization, we work with the log-likelihood, taking the natural logarithm of $L(\mu, \sigma^2)$:

$$\ell(\mu, \sigma^2) = \log L(\mu, \sigma^2) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2.$$

The goal of MLE is to find the parameters μ and σ^2 that maximize $\ell(\mu, \sigma^2)$. This is done by taking partial derivatives with respect to μ and σ^2, setting them to zero, and solving.

1. **Estimate** μ:

$$\frac{\partial \ell}{\partial \mu} = \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu) = 0,$$

$$\sum_{i=1}^{n}(x_i - \mu) = 0,$$

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i,$$

   which is the sample mean.

2. **Estimate** σ^2:

$$\frac{\partial \ell}{\partial \sigma^2} = -\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2}\sum_{i=1}^{n}(x_i - \mu)^2 = 0,$$

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

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

   which is the sample variance (note that this is the biased estimator; an unbiased version uses $n - 1$ in the denominator).

Thus, the MLE for a Gaussian yields $\hat{\mu} = \bar{x}$ (the sample mean) and $\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$ (the sample variance). For multivariate Gaussians in $\mathbb{R}^d$, the approach extends to estimating the mean vector μ and covariance matrix Σ, where:

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i,$$

$$\hat{\Sigma} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})(x_i - \hat{\mu})^T.$$

This MLE framework is critical in clustering, particularly in the EM algorithm for GMMs, where it estimates Gaussian parameters for each cluster during the maximization step.

3.7.5 Expectation Maximization

The Expectation-Maximization (EM) algorithm is an iterative method for finding maximum likelihood estimates of parameters in statistical models with latent variables. It is widely used in clustering, particularly for fitting Gaussian Mixture Models (GMMs), where observed data are assumed to be generated from a mixture of distributions, and cluster assignments are unobserved (latent).

Consider a dataset $X = \{x_1, x_2, \ldots, x_n\}$, where each x_i is generated by a mixture model with K components (e.g., Gaussians). Let $Z = \{z_1, z_2, \ldots, z_n\}$ be the latent variables, where z_i indicates the component (cluster) that x_i belongs to. The parameters θ include the mixture weights π_k, means μ_k, and covariances Σ_k for each component $k = 1, \ldots, K$. The likelihood of the observed data is:

$$p(X|\theta) = \prod_{i=1}^{n} \sum_{k=1}^{K} \pi_k p(x_i|\mu_k, \Sigma_k),$$

where $p(x_i|\mu_k, \Sigma_k)$ is the probability density (e.g., Gaussian) of x_i under component k, and $\sum_{k=1}^{K} \pi_k = 1$, $\pi_k \geq 0$.

Directly maximizing this likelihood is challenging due to the sum inside the product. EM addresses this by iteratively optimizing a lower bound on the log-likelihood using the latent variables.

The EM algorithm alternates between two steps:

1. **E-Step (Expectation)**: Compute the expected value of the log-likelihood with respect to the latent variables Z, given the current parameter estimates $\theta^{(t)}$. This involves calculating the posterior probabilities (responsibilities) of each data point belonging to each component:

$$\gamma_{ik}^{(t)} = p(z_i = k|x_i, \theta^{(t)}) = \frac{\pi_k^{(t)} p(x_i|\mu_k^{(t)}, \Sigma_k^{(t)})}{\sum_{j=1}^{K} \pi_j^{(t)} p(x_i|\mu_j^{(t)}, \Sigma_j^{(t)})},$$

   where $\gamma_{ik}^{(t)}$ represents the probability that x_i belongs to cluster k at iteration t. The expected complete-data log-likelihood, denoted $Q(\theta|\theta^{(t)})$, is then:

$$Q(\theta|\theta^{(t)}) = \sum_{i=1}^{n} \sum_{k=1}^{K} \gamma_{ik}^{(t)} \left[\log \pi_k + \log p(x_i|\mu_k, \Sigma_k)\right].$$

2. **M-Step (Maximization)**: Update the parameters θ to maximize $Q(\theta|\theta^{(t)})$. For a GMM, this yields:

   - Mixture weights: $\pi_k^{(t+1)} = \frac{1}{n} \sum_{i=1}^{n} \gamma_{ik}^{(t)}$,
   - Means: $\mu_k^{(t+1)} = \frac{\sum_{i=1}^{n} \gamma_{ik}^{(t)} x_i}{\sum_{i=1}^{n} \gamma_{ik}^{(t)}}$,
   - Covariances: $\Sigma_k^{(t+1)} = \frac{\sum_{i=1}^{n} \gamma_{ik}^{(t)} (x_i - \mu_k^{(t+1)})(x_i - \mu_k^{(t+1)})^T}{\sum_{i=1}^{n} \gamma_{ik}^{(t)}}$.

These updates are analogous to weighted MLE for Gaussians, with $\gamma_{ik}^{(t)}$ acting as soft weights.

The algorithm iterates between the E-step and M-step until convergence, typically when the log-likelihood $\log p(X|\theta^{(t)})$ or parameters stabilize. EM guarantees that the likelihood does not decrease at each iteration, though it may converge to a local optimum depending on initialization.

EM is powerful for soft clustering (e.g., GMMs), as it naturally provides probabilistic assignments. However, it requires specifying K and can be sensitive to initial parameter values, often necessitating multiple runs or smart initialization (e.g., using K-means). Its time complexity is $O(nK)$ per iteration, making it scalable but slower than hard clustering methods like K-means.

3.7.6 Clustering Properties

- **Richness**: Ensures that all possible partitions of a dataset can be achieved by adjusting the clustering parameters.
- **Scale-Invariance**: Guarantees that the clustering outcome remains unchanged under uniform scaling of the data (e.g., changing units).
- **Consistency**: States that if distances within clusters decrease and between clusters increase, the clustering result should not change.
- **Impossibility Theorem**: Proves that no clustering algorithm can simultaneously satisfy richness, scale-invariance, and consistency for all datasets.

3.8 Feature Selection

Introduction: Feature selection is a critical process in machine learning and data analysis aimed at identifying the **most relevant features** for a given task. It reduces complexity while enhancing model performance and interpretability.

Why Feature Selection?

- **Knowledge Discovery**:
  - *Interpret the Features*: Understand what actually matters in the data.
  - *Insight*: From 1000 features, perhaps only 10 are significant; features we assumed were important may not be.
- **Curse of Dimensionality**: The amount of data required grows exponentially as 2^N, where N is the number of features, making high-dimensional data challenging to work with.

3.8.1 Algorithms

- **Filtering**:
  - *Overview*: This method evaluates and ranks features based on their intrinsic properties, independent of any machine learning model, making it computationally efficient and scalable.
  - *Techniques*: Common approaches include statistical measures like correlation coefficients (e.g., Pearson or Spearman), mutual information, chi-squared tests, or variance thresholds.
  - *Advantages*: Fast execution, no reliance on a specific classifier, and suitability for preprocessing large datasets.
  - *Limitations*: Ignores feature interactions and may select redundant features, as it does not consider model performance directly.
- **Wrapping**:
  - *Overview*: This method uses a specific machine learning model to evaluate subsets of features, selecting those that optimize the model's performance, providing a more tailored feature set.
  - *Techniques*: Employs search strategies like forward selection (start with no features, add one at a time), backward elimination (start with all features, remove one at a time), or recursive feature elimination (RFE) with cross-validation.
  - *Advantages*: Accounts for feature interactions and directly optimizes for the chosen model, often leading to better predictive accuracy.
  - *Limitations*: Computationally expensive due to repeated model training, and results depend heavily on the chosen algorithm and evaluation metric (e.g., accuracy, F1-score).

3.9 Feature Transformation

3.9.1 Principal Components Analysis

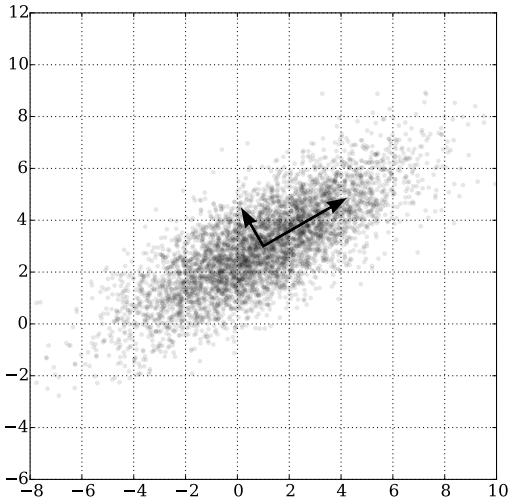


Figure 8: PCA of a multivariate Gaussian centered at (1, 3) with standard deviations of 3 along (0.866, 0.5) and 1 orthogonally yields eigenvectors, scaled by eigenvalue square roots and shifted to the mean, as principal components.

Principal Components Analysis (PCA) is a fundamental feature transformation method that **reduces a dataset's dimensionality by creating new features—principal components—as linear combinations of the originals**. It reorients the data into a coordinate system where axes align with directions of maximum variance, preserving as much information as possible.

How PCA Works: For a dataset X with n observations and p features, PCA begins by standardizing X (zero mean, unit variance) to normalize scales. The covariance matrix $C = \frac{1}{n-1}X^T X$ is then computed, followed by eigen decomposition to extract eigenvalues λ_i (variance magnitudes) and eigenvectors v_i (component directions). Sorting eigenvectors by descending eigenvalues, the top k (where $k < p$) form a matrix W, and the data is projected via $Y = XW$ into a k-dimensional space.

Advantages and Limitations: PCA is efficient, leveraging linear algebra (or SVD for large data), but:

- *Linearity*: It assumes linear relationships, limiting its use for nonlinear data.
- *Variance Focus*: It may not optimize for tasks like classification.
- *Interpretability*: Components lose original meaning.

3.9.2 Independent Component Analysis

Independent Component Analysis (ICA) is a powerful feature transformation method that **separates a multivariate dataset into statistically independent components,**

assuming the data arises from a linear mixture of independent sources. Unlike PCA, which focuses on variance, ICA seeks to uncover hidden, non-Gaussian sources by maximizing their independence, making it ideal for blind source separation tasks.

How ICA Works: For a dataset X with n observations and p features, ICA assumes $X = AS$, where S is a matrix of p independent source signals, and A is an unknown mixing matrix. The goal is to estimate an unmixing matrix W such that $Y = WX \approx S$, recovering the sources. ICA typically preprocesses X by centering (zero mean) and whitening (unit variance, uncorrelated components, often via PCA). Then, it optimizes W by maximizing a measure of independence, such as negentropy or mutual information, often using iterative algorithms like FastICA.

Advantages and Limitations: ICA excels at separating independent signals, but:

- *Linearity*: It assumes the mixing process is linear, limiting its use for nonlinear mixtures.
- *Non-Gaussianity*: It requires sources to be non-Gaussian (at most one Gaussian source is allowed).
- *Ambiguity*: The scale and order of recovered components are indeterminate.

3.10 Cocktail Party Problem

The Cocktail Party Problem is a classic signal processing challenge that **involves separating individual sound sources—such as speakers' voices—from a mixture recorded by multiple sensors, mimicking the human ability to focus on one conversation in a noisy room**. It exemplifies blind source separation, where the goal is to recover independent sources without knowing how they were mixed.

How It Works: Consider p sources (e.g., speakers) emitting signals $S = [s_1, s_2, \ldots, s_p]$, recorded by m microphones as $X = AS$, where A is an unknown mixing matrix capturing how each source contributes to each recording. The task is to estimate an unmixing matrix W such that $Y = WX \approx S$. Independent Component Analysis (ICA) is a primary solution: after preprocessing X (centering and whitening), ICA maximizes the statistical independence of the components in Y, often using measures like negentropy, to recover the original signals.

Advantages and Limitations: This approach effectively isolates independent sources, but:

- *Linearity*: It assumes a linear mixing process, unrealistic for complex acoustics.
- *Source Count*: The number of microphones must typically match or exceed the number of sources ($m \geq p$).
- *Noise*: Real-world noise or reverberation can degrade separation quality.

4 Reinforcement Learning

4.1 Markov Decision Processes

A *Markov Decision Process* (MDP) provides a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of an agent. It is defined by the following components:

- **States** (S): A finite set of states representing all possible situations or configurations the agent can be in. For a given MDP, S encompasses the entire state space, and at any time step, the agent occupies exactly one state $s \in S$.
- **Model** ($T(s, a, s')$): The transition model, often denoted as $T(s, a, s') = P(s'|s, a)$, which specifies the probability of transitioning to state $s' \in S$ given that the agent is in state $s \in S$ and takes action a. This satisfies the Markov property, meaning the future state depends only on the current state and action, not the prior history.
- **Actions** ($A(s)$): A set of possible actions available to the agent in state s. The function $A(s)$ maps each state to its allowable actions, which may vary depending on the state. In some cases, the action set A is the same for all states.
- **Reward** ($R(s)$): The reward function, which assigns a numerical value to each state $s \in S$ (or alternatively to state-action pairs $R(s, a)$ or transitions $R(s, a, s')$). This represents the immediate feedback the agent receives upon entering a state, guiding the learning process toward maximizing cumulative rewards.
- **Policy** ($\pi(s)$): A strategy that defines the agent's behavior by mapping each state $s \in S$ to an action $a \in A(s)$. A policy $\pi(s)$ can be deterministic (specifying a single action) or stochastic (providing a probability distribution over actions). **The goal in reinforcement learning is to find an optimal policy π^* that maximizes the expected cumulative reward.**

An MDP assumes a fully observable environment, where the agent has complete knowledge of its current state. Solving an MDP typically involves computing the expected future rewards (value functions) and optimizing the policy based on these values.

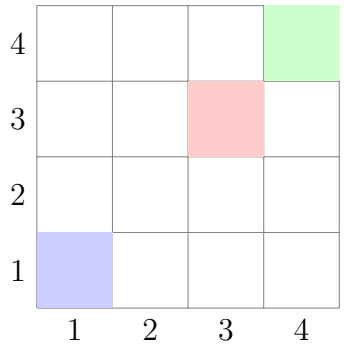
4.2 Rewards

In reinforcement learning, rewards serve as the primary **signal** guiding the agent toward desirable behaviors. The following concepts highlight the complexity and nuances of reward structures:

- **Delayed Rewards**: Rewards are not always immediate; they may be received after a sequence of actions rather than directly following a single decision. This delay complicates learning, as the agent must infer which actions contributed to the eventual outcome.
- **Minor Changes Matter**: Small decisions can have significant long-term impacts. For example, in chess, a seemingly minor opening move might position the player for a win or loss much later in the game, illustrating how early actions can shape future rewards.

- **Temporal Credit Assignment**: This refers to the challenge of determining which actions in a sequence are responsible for a received reward. When rewards are delayed, the(agent must solve this problem to assign credit (or blame) to past decisions, a key issue in reinforcement learning.
- **Stochastic Rewards**: Rewards can be probabilistic rather than deterministic, meaning the same action in the same state may yield different outcomes. This randomness requires the agent to learn expected values over time rather than relying on fixed results.
- **Sequences of Rewards**: The agent's goal is typically to maximize the cumulative reward over a sequence of time steps, often formalized as the expected return $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$, where R_t is the reward at time t and T is the final time step. Discounting (e.g., using a factor $\gamma \in [0, 1]$) may be applied to prioritize earlier rewards: $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.

Grid World Example: Consider a 4x4 grid where an agent starts at position (1,1) and aims to reach a goal at (4,4). The state space S consists of all grid cells (i, j), and actions $A(s)$ include moving up, down, left, or right (with boundaries preventing invalid moves). The reward structure might be: $R(s) = -1$ for each step (encouraging efficiency), except at (4,4), where $R(s) = +10$, and at (3,3), where a stochastic trap gives $R(s) = -5$ with 50% probability and $R(s) = 0$ otherwise. Below is a depiction of this Grid World:



In this grid, delayed rewards occur as the +10 is only received upon reaching (4,4), minor moves (e.g., avoiding (3,3)) matter significantly, and temporal credit assignment is needed to link early moves to the final reward. The stochastic trap introduces uncertainty, and the agent optimizes a sequence of rewards over multiple steps.

These properties underscore the need for algorithms that can handle delayed feedback, uncertainty, and long-term planning, such as those based on dynamic programming, Monte Carlo methods, or temporal-difference learning.

4.3 Sequence of Rewards

In reinforcement learning, the agent aims to maximize total rewards over time. Here's how sequences of rewards work:

- **Infinite Horizons**

  - No fixed end to the task—agent keeps going forever.
  - Return is the sum of all future rewards: $G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots$.
  - To keep this finite, use a discount factor γ (between 0 and 1).
  - Discounted return: $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.
  - γ decides focus: small γ for short-term, big γ for long-term rewards.

- **Utility of Sequences**
  - Utility = value of the reward sequence to the agent.
  - Measured as expected return G_t based on policy $\pi(s)$.
  - Value function $V^\pi(s)$ = expected return starting at state s with policy π.
  - Action-value $Q^\pi(s, a)$ = expected return for action a in state s, then following π.
  - Goal: find the best policy π^* to maximize utility across all states.

These ideas help agents plan for long or endless tasks using methods like value iteration or policy gradients.

4.4 Policies

A policy guides the agent's actions. Here are the math definitions for key terms, including the Bellman equation:

- **Define π^***
  - π^* is the optimal policy.
  - It's the strategy that gives the highest possible rewards across all states.
  - Uses $\pi^*(s)$ to pick the best action for each state s.
  - Goal: maximize long-term value in every situation.

- **Define $\pi^*(s)$**
  - $\pi^*(s)$ is the action chosen by the optimal policy in state s.
  - Math definition: $\pi^*(s) = \arg\max_{a \in A(s)} Q^*(s, a)$.
  - This means: pick the action a from $A(s)$ that maximizes $Q^*(s, a)$.
  - $Q^*(s, a)$ is the best expected return starting from s, taking a, then acting optimally.

- **Define $U^\pi(s)$**
  - $U^\pi(s)$ is the utility of state s under policy π.
  - Math definition: $U^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right]$.
  - This means: expected sum of discounted rewards from state s, following π.
  - γ (0 to 1) discounts future rewards; R_{t+k+1} is each step's reward.

- **Bellman Equation**

- Links utility to rewards and future states.
- Math definition: $U^\pi(s) = R(s) + \gamma \sum_{s' \in S} T(s, \pi(s), s') U^\pi(s')$.
- This means: utility of state s is the immediate reward $R(s)$ plus the discounted expected utility of the next state s'.
- $T(s, \pi(s), s')$ is the probability of moving to s' from s with action $\pi(s)$.
- Used to compute $U^\pi(s)$ and find π^* by evaluating policies.

These formulas, especially the Bellman equation, help the agent calculate values ($U^\pi(s)$) and find the best policy (π^* and $\pi^*(s)$).

4.5 Finding Policies

This section covers how to find policies, like π^*, by solving the Bellman equation:

- **Goal**
  - Find the optimal policy π^* that maximizes rewards.
  - Use the Bellman equation to compute utilities and choose actions.
- **Bellman Equation Recap**
  - For a policy π: $U^\pi(s) = R(s) + \gamma \sum_{s' \in S} T(s, \pi(s), s') U^\pi(s')$.
  - $R(s)$ is the immediate reward, γ discounts future rewards.
  - $T(s, \pi(s), s')$ is the chance of reaching state s' from s with action $\pi(s)$.
- **Solving the Bellman Equation**
  - Step 1: Start with a policy π (can be random).
  - Step 2: For each state s, compute $U^\pi(s)$ using the equation.
  - This is a system of equations—one for each state in S.
  - Example: If 3 states (s_1, s_2, s_3), solve:
    - $U^\pi(s_1) = R(s_1) + \gamma \sum_{s'} T(s_1, \pi(s_1), s') U^\pi(s')$,
    - $U^\pi(s_2) = R(s_2) + \gamma \sum_{s'} T(s_2, \pi(s_2), s') U^\pi(s')$,
    - $U^\pi(s_3) = R(s_3) + \gamma \sum_{s'} T(s_3, \pi(s_3), s') U^\pi(s')$.
  - Step 3: Solve these equations (e.g., with linear algebra) to get $U^\pi(s)$ for all s.
- **Improving the Policy**
  - Once $U^\pi(s)$ is known, update the policy.
  - New policy: $\pi'(s) = \arg\max_{a \in A(s)} \left[R(s) + \gamma \sum_{s' \in S} T(s, a, s') U^\pi(s') \right]$.
  - This picks the action a that maximizes the expected utility.
  - Repeat: Solve Bellman for π', update policy, until π stops changing.
  - Final result: π^* and $U^*(s)$, the optimal policy and utilities.

Solving the Bellman equation this way (called policy iteration) finds π^* step-by-step.

4.6 Three Approaches to RL

Reinforcement learning (RL) has three main ways to find good policies. Here they are:

- **Policy Search (π)**
  - Focus: Directly find the best policy $\pi(s)$.
  - How: Try different policies and tweak them based on rewards.
  - No need for $U(s)$ or a model—just adjust π to maximize return.
  - Example: Use gradients (policy gradient methods) to improve π.
  - Good for: High-dimensional or continuous action spaces.
- **Value-Function Based (U)**
  - Focus: Learn the utility $U(s)$ or $Q(s, a)$ for states or actions.
  - How: Use Bellman equations to estimate values, then pick actions.
  - Policy comes from values: $\pi(s) = \arg\max_a Q(s, a)$.
  - Example: Q-learning or value iteration to compute U or Q.
  - Good for: Problems where values guide decisions well.
- **Model-Based (T, R)**
  - Focus: Build a model of the environment using $T(s, a, s')$ and $R(s)$.
  - How: Learn transition probabilities T and rewards R, then plan.
  - Use the model to simulate and find π^* (e.g., with dynamic programming).
  - Example: Learn T and R from experience, then solve the MDP.
  - Good for: When the environment can be modeled accurately.

Each approach—policy search, value-based, or model-based—suits different RL problems.

4.7 Q-Learning

Q-Learning is a model-free, value-function based method in reinforcement learning. It learns the optimal action-value function $Q^*(s, a)$ to find the best policy π^*.

- **What It Does**
  - Goal: Learn $Q^*(s, a)$, the best expected reward from state s, action a, then acting optimally.
  - Gives $\pi^*(s) = \arg\max_a Q^*(s, a)$ without needing $T(s, a, s')$ or $R(s)$.
- **Update Rule**
  - Formula: $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$.
  - From experience: (s, a, r, s')—current state, action, reward, next state.
  - α (0 to 1): Learning rate, adjusts update size.

  - γ (0 to 1): Discount factor, weights future rewards.
  - $\max_{a'} Q(s', a')$: Best value from s', assuming optimal future actions.

- **Choosing Actions**
  - Policy: During learning, actions aren't just $\pi^*(s)$.
  - Exploration vs. exploitation: Balance trying new actions and using known good ones.
  - Common method: ϵ-greedy (see below).

- **Greedy Exploration**
  - ϵ-greedy: Pick best action ($\arg\max_a Q(s, a)$) with probability $1 - \epsilon$.
  - Random action: Choose any a with probability ϵ (e.g., $\epsilon = 0.1$).
  - Adjust ϵ: Start high (explore more), decrease over time (exploit more).
  - Ensures: Agent tries all actions, avoids missing better options.

- **Learning Convergence**
  - Process: Start with $Q(s, a)$ (e.g., zeros), update after each step.
  - Converges to Q^*: If all state-action pairs are visited infinitely often.
  - Conditions: α must decrease (e.g., $1/t$), sum to infinity, and exploration persists.
  - Result: $Q(s, a) \approx Q^*(s, a)$, so $\pi^*(s)$ is optimal.

- **Key Features**
  - Model-free: No need for a model, learns from experience.
  - Off-policy: Learns Q^* regardless of exploration actions.
  - Practical: Works with tuning (e.g., fixed small α, decaying ϵ).

- **Using It**
  - After learning: $\pi^*(s) = \arg\max_a Q^*(s, a)$.
  - Good for: Discrete spaces like grid worlds.
  - Limits: Needs tweaks for large or continuous spaces.

Q-Learning balances exploration and exploitation to converge to an optimal policy in unknown environments.

4.8 Game Theory