Machine Learning

CS 446 - Taught by Liangyan Gui

Fall 2025 - Notes compiled by Ayush Singh

Abstract

Principles and applications of machine learning. Main paradigms and techniques, including discriminative and generative methods, reinforcement learning: linear regression, logistic regression, support vector machines, deep nets, structured methods, dimensionality reduction, k-means, Gaussian mixtures, expectation maximization, Markov decision processes, and Q-learning.

Contents

Part I

Foundations & Theory

1 Paradigms

1.1 Supervised Learning

Given an input space $\mathcal{X}$ and an output space $\mathcal{Y}$, the goal is to construct a prediction rule $h : \mathcal{X} \to \mathcal{Y}$ that predicts an output y given an input x.

There are two common types of supervised learning tasks:

- **Classification:** The output space $\mathcal{Y}$ consists of discrete labels (e.g., $\mathcal{Y} = \{0, 1\}$ or $\mathcal{Y} = \{\text{Sports, News}\}$).
- **Regression:** The output space $\mathcal{Y}$ is continuous (e.g., $\mathcal{Y} = \mathbb{R}$). The goal is to fit a model that can predict a value on a continuous scale.

1.2 Unsupervised Learning

Given data $X \in \mathcal{X}$ without corresponding labels, the goal is to learn a function $h(X)$ that captures underlying structure.

- **Clustering:** Grouping data points into distinct clusters based on similarity.
- **Density Estimation:** Learning the probability distribution of the data.

1.3 Modeling Approaches: Discriminative vs. Generative

1.3.1 Discriminative Classifiers

Discriminative classifiers try to learn the posterior probability $P(Y|X)$ directly.

- They focus strictly on finding the decision boundary that separates classes.
- **Philosophy:** "Don't solve a harder problem (modeling the world) than you have to (just classification)."
- *Examples:* Logistic Regression, Support Vector Machines (SVM), Perceptron.

1.3.2 Generative Classifiers

Generative classifiers learn how the data is generated by building probabilistic models of each class.

- They learn the class prior $P(Y)$ and the class-conditional density $P(X|Y)$.
- Effectively, they model the joint distribution $P(X, Y) = P(X|Y)P(Y)$.
- Predictions are made using Bayes Rule:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \propto P(X|Y)P(Y)$$

- *Examples:* Naive Bayes, Gaussian Discriminant Analysis (GDA).

2 Learning Theory Fundamentals

2.1 Empirical Risk Minimization

The risk of a predictive function h is given by the expected loss:

$$R(h) = \mathbb{E}_{X,Y}[\ell(h(X), Y)]$$

However, we don't know the true distribution. Instead, we use empirical risk minimization (ERM), which minimizes the average loss over the training data:

$$\min_w \frac{1}{N} \sum_{i=1}^{N} \ell(h_w(x^{(i)}), y^{(i)})$$

The one problem with ERM is that a predictor that perfectly memorizes the training data (e.g., $h(x^{(i)}) = y^{(i)}$ for all training points) achieves zero empirical risk but generalizes poorly to new data.

Regularized Empirical Risk Minimization (RERM)

To prevent overfitting, we add a regularization term that penalizes model complexity:

$$\min_w \frac{1}{N} \sum_{i=1}^{N} \ell(h_w(x^{(i)}), y^{(i)}) + \lambda r(w)$$

where $r(w)$ is a regularizer (e.g., $\|w\|_2^2$ or $\|w\|_1$) and λ is a hyperparameter controlling the strength of regularization.

Common Loss Functions

Classification ($y \in \{-1, +1\}$):

- **Zero-One Loss:** $\mathbb{I}(h_w(x^{(i)}) \neq y^{(i)})$ - Actual classification loss but non-continuous and impractical to optimize.
- **Hinge Loss:** $\max(1 - y^{(i)} h_w(x^{(i)}), 0)$ - Used in SVM.
- **Log Loss:** $\log(1 + e^{-y^{(i)} h_w(x^{(i)})})$ - Used in Logistic Regression.
- **Exponential Loss:** $e^{-y^{(i)} h_w(x^{(i)})}$ - Used in AdaBoost.

Regression ($y \in \mathbb{R}$):

- **Squared Loss:** $(h_w(x^{(i)}) - y^{(i)})^2$ - Estimates mean label, differentiable everywhere, but sensitive to outliers.
- **Absolute Loss:** $|h_w(x^{(i)}) - y^{(i)}|$ - Estimates median label, less sensitive to noise, but not differentiable at 0.
- **Huber Loss:** Combines squared and absolute loss - squared when error is small, absolute when large.

Common Regularizers

L2 Regularization (Ridge): $r(w) = \|w\|_2^2 = \sum_j w_j^2$

- Strictly convex and differentiable
- Produces small but non-zero weights
- Corresponds to Gaussian prior in MAP estimation

L1 Regularization (Lasso): $r(w) = \|w\|_1 = \sum_j |w_j|$

- Convex but not differentiable at 0
- Produces sparse solutions (many weights become exactly 0)
- Corresponds to Laplace prior in MAP estimation
- Useful for feature selection

Elastic Net: Combines L1 and L2: $\alpha\|w\|_1 + (1-\alpha)\|w\|_2^2$

2.2 The Bias-Variance Tradeoff

The expected test error can be decomposed into three components:

$$\mathbb{E}_{x,y,D}[(h_D(x)-y)^2] = \underbrace{\mathbb{E}_{x,D}[(h_D(x)-\bar{h}(x))^2]}_{\text{Variance}} + \underbrace{\mathbb{E}_x[(\bar{h}(x)-\bar{y}(x))^2]}_{\text{Bias}^2} + \underbrace{\mathbb{E}_{x,y}[(\bar{y}(x)-y)^2]}_{\text{Noise}}$$

where:

- h_D is the predictor trained on dataset D
- $\bar{h}(x) = \mathbb{E}_D[h_D(x)]$ is the expected predictor averaged over all possible training sets
- $\bar{y}(x) = \mathbb{E}_{y|x}[Y]$ is the expected label given x

Variance: Measures how much the predictor h_D changes when trained on different datasets. High variance means the model is overfitting to the specific training data and is overly sensitive to the particular sample.

Bias: Measures the inherent error of the model family even with infinite training data. High bias means the model is too simple to capture the true underlying pattern (underfitting).

Noise: The irreducible error due to the stochastic nature of the data. This is independent of the model or algorithm.

The Tradeoff

There is typically a tradeoff between bias and variance:

- **Simple models** (e.g., linear models, low-degree polynomials): High bias, low variance. They underfit the data but are stable across different training sets.
- **Complex models** (e.g., high-degree polynomials, deep trees): Low bias, high variance. They can fit the training data well but are sensitive to the specific training sample.

The optimal model complexity lies at the point where the sum of bias and variance is minimized. This is the "sweet spot" that generalizes best to unseen data.

2.3 Model Selection

Model selection is the process of choosing the best hyperparameters (e.g., regularization strength λ, polynomial degree, number of neighbors in kNN) to achieve good generalization.

Hold-Out Method

Split data into training and validation sets (e.g., 80/20 or 70/30 split):

1. Train on the training split with different hyperparameter values
2. Evaluate performance on the validation split
3. Select the hyperparameter that gives the best validation performance

Drawbacks:

- May not have enough data to afford setting aside a large validation set
- Validation error can be misleading if we get an "unfortunate" split

K-Fold Cross-Validation

A more robust approach that uses all data for both training and validation:

1. Split data into K equal-sized folds (typically $K = 5$ or $K = 10$)
2. For each of the K folds:
   - Train on $K-1$ folds
   - Validate on the remaining fold
3. Average the K validation errors to get the cross-validation error
4. Select the hyperparameter with the lowest cross-validation error

Choosing K:

- **Large K:** More accurate estimate of test error, but higher computational cost and higher variance in validation error
- **Small K:** Less computational cost, more stable validation error, but less accurate estimate of test error
- **Common choice:** $K = 10$

Leave-One-Out (LOO) Cross-Validation: Special case with $K = N$. Train on $N-1$ samples and validate on 1 sample, repeated N times. Provides nearly unbiased estimate but very computationally expensive.

Early Stopping

An alternative regularization technique for iterative algorithms (e.g., gradient descent):

- Monitor both training and validation error during training
- Stop training when validation error starts to increase (even if training error is still decreasing)
- This prevents overfitting by limiting the number of optimization steps

Early stopping acts as an implicit regularizer by constraining how far the parameters can move from their initialization.

2.4 Regularization (L1 & L2)

Regularization constrains the model parameters to prevent overfitting. The two most common forms are L1 and L2 regularization.

L2 Regularization (Ridge)

Adds the squared L2 norm of the weights to the loss:

$$\min_w \frac{1}{N}\sum_{i=1}^{N} \ell(h_w(x^{(i)}), y^{(i)}) + \lambda\|w\|_2^2$$

Properties:

- Encourages small weights but doesn't force them to exactly zero
- Has a closed-form solution for linear regression: $w = (X^T X + \lambda I)^{-1} X^T y$
- Equivalent to MAP estimation with a Gaussian prior: $w \sim \mathcal{N}(0, \sigma^2 I)$
- Geometrically, constrains weights to lie within a sphere: $\|w\|_2^2 \leq B$

L1 Regularization (Lasso)

Adds the L1 norm of the weights to the loss:

$$\min_w \frac{1}{N}\sum_{i=1}^{N} \ell(h_w(x^{(i)}), y^{(i)}) + \lambda\|w\|_1$$

Properties:

- Encourages sparse solutions where many weights become exactly zero
- Useful for feature selection - automatically identifies important features
- No closed-form solution - requires iterative optimization
- Equivalent to MAP estimation with a Laplace prior: $p(w_j) \propto e^{-|w_j|/b}$
- Geometrically, constrains weights to lie within a diamond: $\|w\|_1 \leq B$

Geometric Intuition

The sparsity-inducing property of L1 comes from its geometric shape. When the regularization constraint intersects the loss contours, L2 (sphere) typically intersects at a point where all coordinates are non-zero, while L1 (diamond) tends to intersect at corners where many coordinates are exactly zero.

Connection to Priors

Regularization in the optimization framework is equivalent to MAP estimation in the probabilistic framework:

- L2 regularization $\Leftrightarrow$ Gaussian prior
- L1 regularization $\Leftrightarrow$ Laplace prior

Part II

Supervised Learning

3 Probabilistic Models

3.1 Naive Bayes

Naive Bayes is a classification algorithm. It is motivated by the difficulty of trying to learn a probability distribution where there are many features. This results in 2^d parameters that need to be learned, which is impossible due to data scarcity. For example, with just 30 binary features, there are already more combinations of parameters than humans on Earth.

Using Bayes rule, we can try to reduce the number of parameters, but it does not work directly without assumptions. Naive Bayes makes the assumption of conditional independence, where each feature is independent of the others given the class label. This reduces the complexity to $2d$ parameters, which is feasible. These parameters are estimated using Maximum Likelihood Estimation (MLE) or Maximum A Posteriori (MAP).

Naive Bayes is formally derived as:

$$\begin{aligned} P(Y = c|X_1, ..., X_d) &= \frac{P(Y = c)P(X_1, ..., X_d|Y = c)}{P(X_1, ..., X_d)} \\ &\propto P(Y = c)P(X_1, ..., X_d|Y = c) \\ &\propto P(Y = c)\prod_{j=1}^{d} P(X_j|Y = c) \end{aligned}$$

Here, Bayes Rule is implemented with the denominator removed as it is constant across classes; the value outputted is proportional to the posterior probability. This means the output is a score rather than a strictly normalized probability.

Classification Rule

After we calculate scores for every class, we assign the data point to the class with the highest score:

$$Y^{new} \leftarrow \underset{c}{\operatorname{argmax}}\, P(Y = c)\prod_{j=1}^{d} P(X_j^{new}|Y = c)$$

Implementation Note (Log-Sum Trick): Multiplying many small probabilities often results in numerical underflow (rounding to 0). In practice, we maximize the log-posterior instead, turning products into sums:

$$Y^{new} \leftarrow \underset{c}{\operatorname{argmax}} \left[\log P(Y = c) + \sum_{j=1}^{d} \log P(X_j^{new}|Y = c)\right]$$

Parameter Estimation

We estimate our parameters using the following MLEs:

$$\hat{P}^{MLE}(Y = c) = \frac{\text{\# of samples with label } c}{\text{\# of samples}}$$

$$\hat{P}^{MLE}(X_j = k | Y = c) = \frac{\text{\# of samples with label } c \text{ that have feature } X_j \text{ with value } k}{\text{\# of samples with label } c}$$

Problems & Solutions

The Independence Assumption: Naive Bayes assumes conditional independence, but real data is rarely truly independent. Despite this, Naive Bayes often performs remarkably well in practice.

The Zero Probability Problem: There is a chance that for some feature value k, $\hat{P}^{MLE}(X_j = k|Y = c)$ is zero because it was never observed in the training data. This single zero ruins the entire likelihood product. We can work around this problem by using MAP estimates, also known as Laplace smoothing, which adds "hallucinated" examples:

$$\hat{P}^{MAP}(X_j = k|Y = c) = \frac{\sum_{i=1}^{N} \mathbb{I}\{x_j^{(i)} = k \cap y^{(i)} = c\} + l}{\sum_{i=1}^{N} \mathbb{I}\{y^{(i)} = c\} + lK_j}$$

Where:

- l is the smoothing parameter (usually $l = 1$ for Laplace smoothing).
- K_j is the number of distinct values feature j can take (the domain size of feature j).

3.2 Gaussian Naive Bayes

Gaussian Naive Bayes is a variation of Naive Bayes designed for continuous input features $x \in \mathbb{R}$.

Motivation

In the discrete setting, we count occurrences to estimate probabilities. However, for continuous random variables, the probability of observing a specific value is mathematically zero ($P(X = x) = 0$). Therefore, we cannot use frequency counts. Instead, we assume the likelihood follows a specific PDF, typically a Gaussian distribution.

The Gaussian Assumption

We assume that the likelihood of feature j given class c is:

$$P(X_j = x|Y = c) = \mathcal{N}(x; \mu_{jc}, \sigma_{jc}^2) = \frac{1}{\sqrt{2\pi\sigma_{jc}^2}} \exp\left(-\frac{(x - \mu_{jc})^2}{2\sigma_{jc}^2}\right)$$

Parameter Estimation (MLE)

To train this model, we must estimate the mean μ and variance σ^2 for every feature j and every class c.

$$\hat{\mu}_{jc} = \frac{\sum_{i=1}^{N} x_j^{(i)} \mathbb{I}\{y^{(i)} = c\}}{\sum_{i=1}^{N} \mathbb{I}\{y^{(i)} = c\}}$$

$$\hat{\sigma}_{jc}^2 = \frac{\sum_{i=1}^{N} (x_j^{(i)} - \hat{\mu}_{jc})^2 \mathbb{I}\{y^{(i)} = c\}}{\sum_{i=1}^{N} \mathbb{I}\{y^{(i)} = c\}}$$

Variance Constraints & Decision Boundaries

The complexity of the decision boundary depends on the assumptions we make about the variance parameters:

1. **Class-Independent Variance ($\sigma_{jc}^2 = \sigma_j^2$):** We assume the variance is the same for all classes. This results in a linear decision boundary (similar to Logistic Regression).
2. **Isotropic Variance ($\sigma_{jc}^2 = \sigma^2$):** We assume variance is the same for all features and classes. This is the simplest model (spherical clusters).
3. **Full Independence (σ_{jc}^2):** Each class and feature has its own variance. This allows for complex quadratic decision boundaries.

4 Linear & Geometric Models

4.1 Linear Regression

Linear regression is a technique to learn a simple relationship between input features and a continuous outcome. We can then use the relationship that we learn to predict future outcomes.

Probabilistic Interpretation

We assume that the data is generated by a linear predictor with additive Gaussian noise:

$$y = w^T x + \epsilon, \quad \text{where } \epsilon \sim \mathcal{N}(0, \sigma^2)$$

This implies that the conditional distribution of the labels is:

$$p(y|x, w) = \mathcal{N}(w^T x, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - w^T x)^2}{2\sigma^2}\right)$$

In this predictor, ϵ is a residual noise term. It adds randomness to account for the fact that a strictly linear function cannot capture all the nuance in the real world.

From Likelihood to Least Squares

We can derive the objective function by finding the Maximum Likelihood Estimator (MLE). The likelihood of the entire dataset $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$ is the product of the individual probabilities:

$$L(w) = \prod_{i=1}^{N} p(y^{(i)}|x^{(i)}, w)$$

Maximizing the log-likelihood ($\log L(w)$) is mathematically equivalent to minimizing the negative log-likelihood. Since the Gaussian exponential term contains the squared error, this reduces to minimizing the Sum of Squared Errors (SSE):

$$\hat{w}_{MLE} = \underset{w}{\operatorname{argmin}} \sum_{i=1}^{N} (y^{(i)} - w^T x^{(i)})^2$$

Matrix Notation & The Normal Equation

To solve this efficiently, we stack our inputs and outputs into matrices:

$$y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{bmatrix} \in \mathbb{R}^N, \quad X = \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(N)})^T \end{bmatrix} \in \mathbb{R}^{N \times d}$$

The Mean Squared Error (MSE) loss function can then be written compactly as:

$$\ell(w) = \frac{1}{N}||y - Xw||^2 = \frac{1}{N}(y - Xw)^T(y - Xw)$$

To find the optimal w, we take the gradient $\nabla_w \ell(w)$, set it to 0, and solve. This yields the Normal Equation (closed-form solution):

$$w = (X^T X)^{-1} X^T y$$

4.2 The Perceptron

The Perceptron is one of the earliest linear classification algorithms. It tries to find a separating hyperplane that divides the data into two classes. Unlike linear regression which outputs a continuous value, the perceptron outputs a hard binary label $\{-1, 1\}$.

The hypothesis function is:

$$h(x) = \text{sign}(w^T x)$$

If $w^T x > 0$, we predict $+1$. If $w^T x < 0$, we predict -1.

The Perceptron Algorithm

The algorithm is an iterative process. We start with a weight vector w of all zeros. We then loop through the training data one by one. If the model makes a correct prediction, we do nothing. If it makes a mistake, we update the weights to push the boundary in the right direction.

The update rule when a mistake occurs on example $(x^{(i)}, y^{(i)})$ is:

$$w \leftarrow w + y^{(i)} x^{(i)}$$

This works because adding $y^{(i)} x^{(i)}$ rotates the weight vector towards the correct class, making it more likely to classify that point correctly next time.

Convergence

There is a catch. The perceptron is only guaranteed to converge if the data is *linearly separable*. This means there must exist some perfect hyperplane that separates the positives from the negatives with a non-zero margin. If the data is not separable (like the XOR problem), the perceptron will loop forever and never settle on a solution.

4.3 Logistic Regression

Logistic regression is a discriminative classifier that attempts to learn the probability distribution of $P(Y|X)$ directly, as opposed to the Naive Bayes strategy of learning $P(Y)$ and $P(X|Y)$. It does this by fitting a logistic function to the data and then uses Maximum Conditional Likelihood Estimation (MCLE) to check how likely the data is to have occurred if the curve we are fitting is true. We do this with the bias w_0 and the weight matrix w.

The Probabilistic Model

We start by defining a probabilistic model for binary classification based on the sigmoid function. For each input $x \in \mathbb{R}^d$:

$$P(Y = 1|X) = \sigma(w_0 + w^T X) = \frac{1}{1 + e^{-(w_0 + w^T X)}}$$

This is a modified sigmoid function where the weights determine how much each input in X affects the curve. We can write this compactly as $\hat{p}(X; w)$.

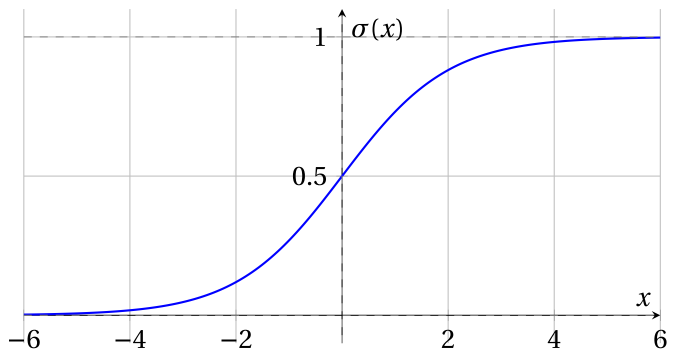

Figure 1: The sigmoid (logistic) function

Likelihood & Optimization

From there, we can find the likelihood of the data. This works because we assume our data is i.i.d (independent and identically distributed):

$$L(w) = \prod_{i=1}^{N} P(y^{(i)}|x^{(i)}, w)$$

For binary labels, we can write the individual probability as:

$$P(y^{(i)}|x^{(i)}, w) = [\hat{p}^{(i)}]^{y^{(i)}}[1 - \hat{p}^{(i)}]^{1-y^{(i)}}$$

Therefore, the full likelihood is:

$$L(w) = \prod_{i=1}^{N} [\hat{p}^{(i)}]^{y^{(i)}}[1 - \hat{p}^{(i)}]^{1-y^{(i)}}$$

We then take the log-likelihood, expressed as $\ell(w) = \log L(w)$. This is critical for the MCLE because it turns products into sums. Our log-likelihood can be written simply as:

$$\ell(w) = \sum_{i=1}^{N} [y^{(i)} z^{(i)} - \log(1 + e^{z^{(i)}})]$$

Here, $z = w_0 + w^T X$. This form is a smooth function which can be differentiated. The log-likelihood function is also concave, meaning there is one unique global maximum. However, there is no closed form solution, meaning we need to use iterative optimization such as gradient methods or Newton's method.

Gradient Update Rule

In order to optimize, we take the gradient of the log-likelihood:

$$\frac{\partial \ell(w)}{\partial w_j} = \sum_i x_j^{(i)}(y^{(i)} - \sigma(z^{(i)}))$$

From there, we update our weights according to the gradient:

$$w_j \leftarrow w_j + \alpha \sum_i x_j^{(i)}(y^{(i)} - \sigma(z^{(i)}))$$

Here, α is the learning rate and $y^{(i)} - \sigma(z^{(i)})$ represents the prediction error for each sample.

Regularization (MAP)

To prevent overfitting, we use a Maximum A Posteriori (MAP) estimate. Our prior is defined as $w \sim \mathcal{N}(0, \sigma^2 I)$. We maximize the posterior $\log P(w) + \ell(w)$. This adjusts our gradient update to include a penalty term (weight decay):

$$w_j \leftarrow w_j + \alpha \Big[\sum_i x_j^{(i)} (y^{(i)} - \sigma(z^{(i)})) - \lambda w_j \Big]$$

Decision Boundary

Once everything has been trained, we can define our decision boundary using the weights we have learned. The boundary exists where the probability is 0.5:

$$\sigma(w_0 + w^T X) = 0.5 \implies w_0 + w^T X = 0$$

4.4 Support Vector Machines (SVM)

SVM is a classification algorithm that works by maximizing the distance, or the margin, between the separating hyperplane and the closest data points.

Geometric Definitions

We use specific terminology to describe the geometry:

- **Median:** The decision boundary itself (the hyperplane).
- **Margin:** The distance from the median to the closest data points on either side.
- **Street:** The region separating the classes, bounded by the margins. We want to maximize the width of this street.
- **Gutters:** The parallel hyperplanes that define the edges of the street.

The hyperplane (median) is defined by the equation:

$$w^T x + b = 0$$

where w is a vector perpendicular to the median. Our decision rule for an unknown sample u is:

$$h(u) = \text{sign}(w^T u + b)$$

For all training points, we want them to be outside the street, correctly classified:

$$y^{(i)}(w^T x^{(i)} + b) \geq a$$

Here, a is a scaling factor. To make the math easier, we arbitrarily set $a = 1$. This defines our "canonical" hyperplanes at the gutters:

$$y^{(i)}(w^T x^{(i)} + b) \geq 1$$

Points strictly inside the gutters (= 1) are called *Support Vectors.*

Maximizing the Margin

We need to find the distance between the two gutters. Let x_+ be a point on the positive gutter ($w^T x_+ + b = 1$) and x_- be a point on the negative gutter ($w^T x_- + b = -1$).

The width of the street is the projection of the vector $(x_+ - x_-)$ onto the unit normal vector $\frac{w}{||w||}$:

$$\text{Width} = (x_+ - x_-) \cdot \frac{w}{||w||}$$

From our gutter equations, we know $w^T(x_+ - x_-) = 2$. Substituting this in:

$$\text{Width} = \frac{2}{||w||}$$

To maximize the width $\frac{2}{||w||}$, we must minimize the length of the weight vector $||w||$. Mathematically, it is equivalent (and easier) to minimize the squared norm:

$$\min_w \frac{1}{2}||w||^2 \quad \text{subject to} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1$$

4.5 Kernel Methods

Kernel methods are a way to resolve the bias-variance tradeoff. If a linear model has high bias (underfitting) because the data isn't linearly separable, we can map the input features into a higher-dimensional space $\phi(x)$. However, explicitly computing $\phi(x)$ is often computationally expensive or impossible if the dimensions are infinite.

The kernel trick allows us to learn in this high-dimensional space without ever computing the coordinates $\phi(x)$ or the weight vector w explicitly.

Before defining the kernel, we must prove that we can even solve the problem using only dot products. It can be proved via induction that for algorithms like gradient descent on squared loss, the weight vector w always lies in the linear span of the training data:

$$w = \sum_{i=1}^{N} \alpha^{(i)} x^{(i)}$$

This means that our hypothesis $h(x) = w^T x$ can be rewritten entirely in terms of inner products between data points:

$$h(x) = \left(\sum_{i=1}^{N} \alpha^{(i)} x^{(i)} \right)^T x = \sum_{i=1}^{N} \alpha^{(i)} (x^{(i)})^T x$$

Since the model only relies on the dot product $(x^{(i)})^T x$, we can replace this linear dot product with a generalized kernel function $k(x^{(i)}, x)$ that computes the dot product in a higher dimensional feature space.

The Kernel Trick

A kernel function computes the inner product in the projected space $\phi(x)$ using only operations in the original space.

$$k(x, z) = \phi(x)^T \phi(z)$$

This provides a massive computational advantage. For example, explicitly computing a polynomial expansion of degree d takes $O(d^p)$ or even $O(2^d)$ time, but calculating the kernel function only takes $O(d)$ time.

We often pre-compute these values for the training set and store them in the Gram Matrix (or Kernel Matrix) K, where $K_{ij} = k(x^{(i)}, x^{(j)})$.

Common Kernels

- **Linear Kernel:** $k(x,z) = x^T z$. This is equivalent to standard linear learning.
- **Polynomial Kernel:** $k(x,z) = (1 + x^T z)^p$. This maps data into a space capturing interactions up to degree p.
- **Radial Basis Function (RBF) / Gaussian Kernel:**
$$k(x,z) = \exp\left(-\frac{||x-z||^2}{2\sigma^2}\right)$$
This is a universal approximator. It corresponds to a feature vector in an infinite-dimensional space.

Valid Kernels & Mercer's Theorem

Not every function can be a kernel. A function $k(x,z)$ is a valid kernel if and only if the corresponding Gram matrix K is positive semi-definite (PSD) for any set of data points. This is known as Mercer's Theorem.

We can also construct new valid kernels from existing ones using specific rules:

- **Scaling:** $ck(x,z)$ for $c > 0$.
- **Sum:** $k_1(x,z) + k_2(x,z)$.
- **Product:** $k_1(x,z)k_2(x,z)$.
- **Exponentiation:** $\exp(k(x,z))$.

Kernel Linear Regression

We can apply this logic to regression. The dual solution for the weights α (where $w = \sum \alpha_i x^{(i)}$) is given by:
$$\alpha = K^{-1}y$$
To make a prediction on a new point x, we compute a weighted sum of the similarities between the new point and the training points:
$$h(x) = \sum_{i=1}^{N} \alpha_i k(x, x^{(i)})$$

5 Instance-Based & Non-Linear Models

5.1 K-Nearest Neighbors (kNN)

kNN works under the assumption of local smoothness: points near each other likely share similar labels. Unlike other algorithms that learn a fixed set of parameters (like w), kNN is a "lazy learner"—it simply memorizes the training data and performs computation only when a prediction is requested.

The Algorithm

Given a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$ and a new query point x:

1. Compute the distance between x and every point in the training set.
2. Select the K points with the smallest distances.
3. **Classification:** Take a majority vote of the labels of these K neighbors.
4. **Regression:** Take the average of the values of these K neighbors.

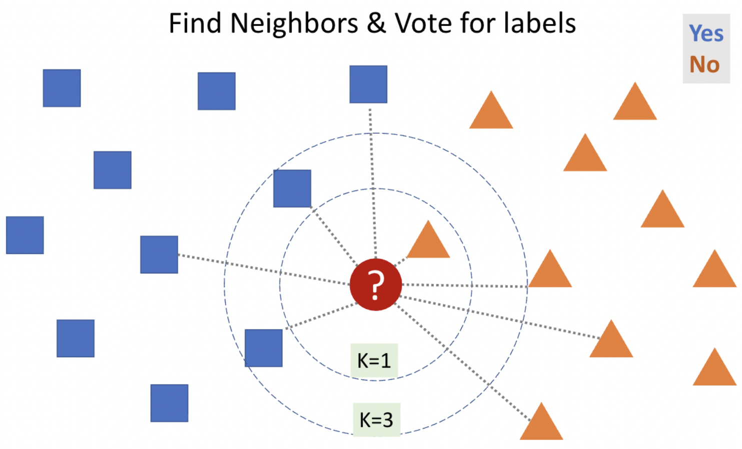

Figure 2: Visualizing kNN Voting: The query point "?" is classified based on the majority label of its neighbors. At $K = 1$, it copies the nearest neighbor. At $K = 3$, it takes a vote.

Distance Metrics

To define "nearest," we need a distance metric. kNN typically uses the Minkowski Distance (Lp norm):

$$D(x,z) = \left(\sum_{d=1}^{D} |x_d - z_d|^p \right)^{1/p}$$

- $p = 1$: Manhattan Distance (L1). Best for high-dimensional, sparse data.
- $p = 2$: Euclidean Distance (L2). The standard geometric distance.
- $p = \infty$: Chebyshev Distance (L∞). The maximum difference along any single dimension.

Choosing K: The Bias-Variance Tradeoff

The choice of K is the critical hyperparameter that controls model complexity.

- **Low K (e.g., K=1):** The model is highly flexible and jagged. It captures local patterns perfectly but is prone to overfitting (High Variance) and sensitive to noise.
- **High K:** The decision boundary becomes smoother. It suppresses noise but ignores local structure, leading to underfitting (High Bias). As $K \to N$, the model simply predicts the majority class of the entire dataset.

Issues with kNN

- **Computational Cost:** Training is free $O(1)$, but testing is expensive $O(N)$ because we must compute distance to every training point.
- **Storage:** Must store the entire training dataset in memory.
- **The Curse of Dimensionality:** In high dimensions, "nearest" neighbors become meaningless.
  - As dimensions increase, the volume of the space grows exponentially.
  - Data becomes incredibly sparse; to capture a "local" neighborhood, we have to encompass a huge volume of the space, effectively including far-away points.

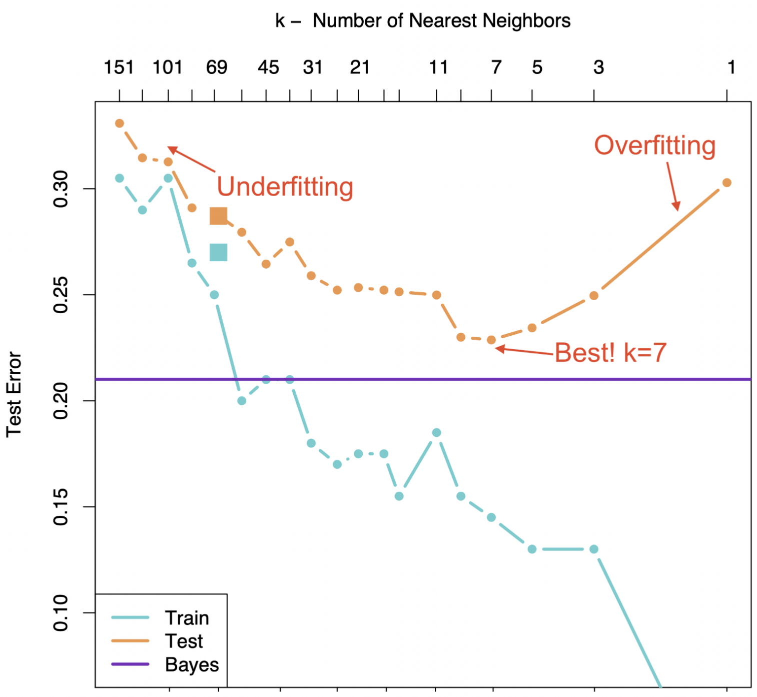

Figure 3: The Bias-Variance Tradeoff in kNN. Small K (right side) leads to overfitting (high variance). Large K (left side) leads to underfitting (high bias). The optimal K lies in the "sweet spot" in the middle.

  – In very high dimensions, the distance to the nearest neighbor approaches the distance to the farthest neighbor.

5.2 Decision Trees

6 Deep Learning

6.1 Neural Networks

Neural networks are computational models inspired by biological neurons. They learn hierarchical representations of data through multiple layers of nonlinear transformations.

Biological Neurons:

- Receive signals through dendrites
- Sum weighted inputs at the cell body
- Apply a threshold function
- Output a signal through the axon if the threshold is exceeded
- Connect to other neurons via synapses

Key Insight: Simple computational units, when connected in networks, can perform complex computations.

An artificial neuron computes a weighted sum of inputs followed by an activation function:

$$z = \sigma\left(\sum_{j=1}^{d} w_j x_j + w_0\right) = \sigma(w^T x)$$

where:

- x_j are input features
- w_j are weights
- w_0 is the bias term
- $\sigma(\cdot)$ is the activation function
- z is the output

We can absorb the bias into the input:

$$x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}, \quad w = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}$$

Then:

$$z = \sigma(w^T x)$$

Activation functions introduce nonlinearity into the network.

6.1.1 Threshold Function

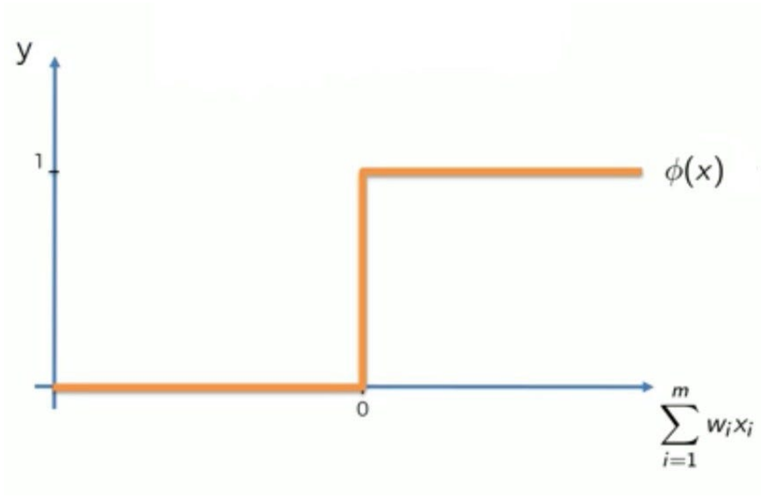

Figure 4: A threshold function.

$$\sigma(a) = \begin{cases} 1 & a \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- Used in the original perceptron
- Non-differentiable at $a = 0$
- Binary output

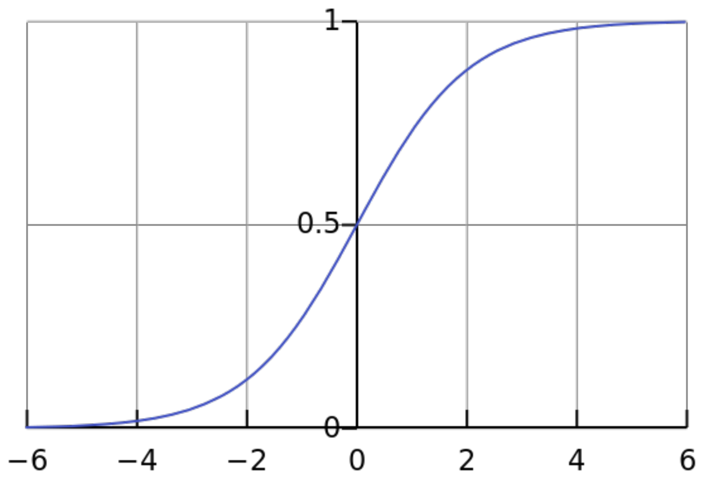

Figure 5: A sigmoid curve.

6.1.2 Sigmoid

$$\sigma(a) = \frac{1}{1+e^{-a}}$$
$$\sigma'(a) = \sigma(a)(1-\sigma(a))$$

- Output range $[0,1]$
- Smooth and differentiable
- Suffers from vanishing gradients

6.1.3 Tanh

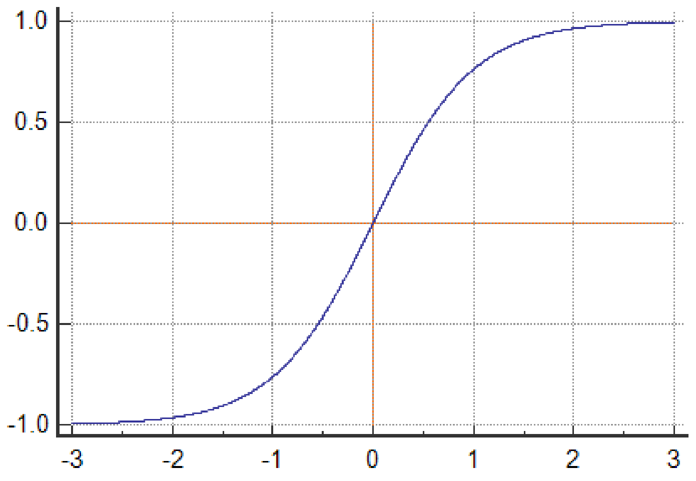

Figure 6: A hyperbolic tangent function.

$$\sigma(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

- Output range $[-1,1]$

- Zero-centered
- Still suffers from vanishing gradients

6.1.4 ReLU

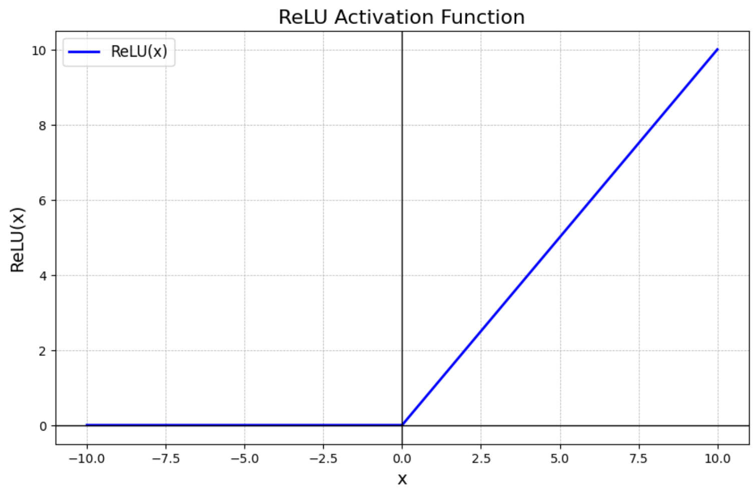

Figure 7: A ReLU function.

$$\sigma(a) = \max(0, a)$$

- Most commonly used activation
- Derivative is 1 for $a > 0$, 0 otherwise
- Efficient and reduces vanishing gradients

Without nonlinear activations:

$$f(x) = W_2(W_1 x) = (W_2 W_1)x$$

This collapses to a linear function regardless of depth. Nonlinear activations enable learning complex patterns.

For K hidden units:

$$h(x) = \sum_{i=1}^{K} a_i \sigma(w_i^T x) + b$$

With ReLU:

$$h(x) = \sum_{i=1}^{K} a_i \max(0, w_i^T x) + b$$

This produces a piecewise linear function.

A neural network with:

- One hidden layer
- Sufficient neurons
- Non-polynomial activation

can approximate any continuous function

$$f : \mathbb{R}^d \to \mathbb{R}^M$$

on compact subsets of $\mathbb{R}^d$.

This guarantees representational power, not learnability or generalization.

6.1.5 Loss Function

For one example:

$$e = \frac{1}{2}\|z^* - z\|^2$$

For the dataset:

$$E = \sum_{i=1}^{N} \frac{1}{2}\|z^{*(i)} - z^{(i)}\|^2$$

6.2 Backpropagation

Backpropagation applies the chain rule efficiently.

Forward Pass:

- Compute activations
- Compute loss

Backward Pass:

- Compute gradients layer by layer
- Update weights

For squared loss:

$$\frac{\partial e}{\partial z_A} = z_A - z_A^*$$

For sigmoid:

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

Deep networks learn hierarchical features:

- Early layers: edges and colors
- Middle layers: textures and shapes
- Late layers: objects and semantics

They are more parameter-efficient and generalize better than shallow networks.

- Vanishing gradients: use ReLU
- Exploding gradients: use clipping
- Overfitting: dropout, regularization
- Slow convergence: better optimizers, initialization

6.3 Convolutional Neural Networks

CNNs exploit spatial structure using local connectivity and weight sharing.

Input:

$$C_{in} \times H \times W$$

Filter:

$$C_{in} \times k \times k$$

Output:

$$K \times H' \times W'$$

Parameter count:

$$C_{in} \cdot k \cdot k \cdot K$$

Max Pooling: takes maximum in region **Average Pooling**: takes mean
Pooling reduces spatial resolution and improves invariance.
For fully connected layers:

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{ij}, \quad \sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{ij}-\mu_j)^2$$

$$\hat{x}_{ij} = \frac{x_{ij}-\mu_j}{\sqrt{\sigma_j^2+\epsilon}}$$

Skip connection:

$$y = F(x) + x$$

This enables very deep networks by easing gradient flow.

Part III

Unsupervised Learning

6.4 Clustering

Clustering is an unsupervised learning task that groups data points so that points in the same cluster are similar and points in different clusters are dissimilar.

6.4.1 K-means Clustering

K-means partitions the dataset into K clusters by minimizing within-cluster squared distances.

Objective. Let $\mu_k \in \mathbb{R}^d$ be the centroid of cluster k, and let $c(i) \in \{1, \dots, K\}$ be the cluster assignment for point x_i. The K-means objective is

$$J = \sum_{i=1}^{N} \|x_i - \mu_{c(i)}\|_2^2.$$

Algorithm (Lloyd's algorithm).

1. Initialize centroids $\mu_1, \dots, \mu_K$ (commonly K-means++).
2. Repeat until convergence:
   - **Assignment step:**
   $$c(i) = \arg\min_{k \in \{1,\dots,K\}} \|x_i - \mu_k\|_2^2.$$
   - **Update step:**
   $$\mu_k = \frac{1}{|\mathscr{C}_k|} \sum_{i:c(i)=k} x_i, \quad \mathscr{C}_k = \{i : c(i) = k\}.$$

Notes.

- K-means assumes roughly spherical clusters and uses Euclidean distance.
- It can converge to a local optimum, so multiple restarts are common.
- Choosing K is typically done with an elbow plot, silhouette score, or validation on a downstream task.

6.4.2 Hierarchical Clustering

Hierarchical clustering builds a hierarchy of clusters.

Agglomerative clustering. Start with each point as its own cluster, then repeatedly merge the closest clusters until one cluster remains.

Linkage criteria. For clusters A and B:

- **Single linkage:** $\min_{x \in A, y \in B} \|x - y\|$
- **Complete linkage:** $\max_{x \in A, y \in B} \|x - y\|$
- **Average linkage:** average pairwise distance
- **Ward linkage:** merge that minimizes increase in within-cluster variance

The result is often visualized with a **dendrogram**. To obtain K clusters, cut the dendrogram at an appropriate height.

6.4.3 Gaussian Mixture Models (GMMs)

A GMM models data as coming from a mixture of Gaussians:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x \mid \mu_k, \Sigma_k), \quad \pi_k \geq 0, \quad \sum_{k=1}^{K} \pi_k = 1.$$

Soft assignments. GMMs assign probabilities of cluster membership:

$$\gamma_{ik} = p(z_i = k \mid x_i).$$

EM algorithm.

- **E-step:** compute responsibilities γ_{ik} given current parameters
- **M-step:** update $\{\pi_k, \mu_k, \Sigma_k\}$ using γ_{ik}

Notes.

- K-means is a special case of a GMM with equal spherical covariances and hard assignments.
- GMMs can model elliptical clusters via Σ_k.

6.4.4 Cluster Evaluation (Common Metrics)

Inertia (K-means objective).

$$J = \sum_{i=1}^{N} \|x_i - \mu_{c(i)}\|_2^2.$$

Silhouette score. Let $a(i)$ be the average distance from x_i to points in its cluster, and let $b(i)$ be the minimum average distance from x_i to points in any other cluster. Then

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}, \quad -1 \leq s(i) \leq 1.$$

Higher is better.

Practical note. If clustering is used as a preprocessing step for a supervised task, the best choice is often the one that improves downstream validation performance.

7 Dimensionality Reduction

7.1 Principal Component Analysis (PCA)

PCA is a linear dimensionality reduction method that finds directions of maximum variance in the data and projects onto a lower-dimensional subspace.

7.1.1 Setup and Notation

Let the dataset be $X \in \mathbb{R}^{N\times d}$ where N is the number of examples and d is the feature dimension. Let $x_i \in \mathbb{R}^d$ denote the ith row of X.

Centering the data:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad \tilde{x}_i = x_i - \mu$$

Let $\tilde{X} \in \mathbb{R}^{N\times d}$ be the centered data matrix.

7.1.2 Covariance Matrix and Objective

The sample covariance matrix is

$$S = \frac{1}{N}\tilde{X}^T\tilde{X} \in \mathbb{R}^{d\times d}.$$

The first principal component v_1 is the unit vector that maximizes the variance of the projected data:

$$v_1 = \arg\max_{\|v\|_2=1} v^T S v.$$

The solution is the eigenvector of S with the largest eigenvalue.

For the top k principal components, let $V_k = [v_1, \dots, v_k] \in \mathbb{R}^{d\times k}$ be the matrix of the top k eigenvectors. The low-dimensional representation is

$$Z = \tilde{X}V_k \in \mathbb{R}^{N\times k}.$$

7.1.3 Reconstruction and Error

A rank-k reconstruction of the centered data is

$$\hat{X} = ZV_k^T = \tilde{X}V_kV_k^T.$$

PCA can also be derived as minimizing reconstruction error:

$$V_k = \arg\min_{V\in\mathbb{R}^{d\times k},\, V^TV=I} \|\tilde{X} - \tilde{X}VV^T\|_F^2.$$

7.1.4 SVD View

Using the singular value decomposition:

$$\tilde{X} = U\Sigma V^T,$$

the columns of V are the principal directions. The top k columns of V give V_k.

7.1.5 Explained Variance

Let $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$ be the eigenvalues of S. The fraction of variance explained by the first k components is

$$\mathrm{EVR}(k) = \frac{\sum_{j=1}^{k}\lambda_j}{\sum_{j=1}^{d}\lambda_j}.$$

7.1.6 Practical Notes

- Always center features before PCA.
- Consider standardizing features (z-score) if they have different units.
- Choose k via explained variance or validation performance on the downstream task.