Scientific Machine Learning 10
K-Nearest Neighbor and Feature Scaling

Donghyun Ko

January 4, 2026

What you will learn in this post. You'll get a clear, practical tour of K-Nearest Neighbors (KNN) as a *lazy*, non-parametric learner: how it predicts (classification by majority vote, regression by averaging), how to choose K (bias–variance trade-off, tie handling, optional distance-based weighting), and why distance choice and *feature scaling* govern performance. You'll compare key metrics—Euclidean/Manhattan/Minkowski, cosine (for directional/sparse data), Hamming (binary/categorical), Jaccard (sets), and Mahalanobis (correlated features)—and learn when each fits your data. You'll see why scaling is essential for distance- and dot-product-based models, when it's optional (tree rules, NB/LDA), and how to do it correctly: min–max, z-score standardization, robust median/IQR, max–abs (preserve sparsity), and unit-length—always *fit on training, apply to validation/test* to avoid leakage. We'll cover data prep (handle missingness, reduce dimensionality), the *curse of dimensionality* and its remedies, practical tips for matching scaling to data and model, and computational/memory implications that arise because KNN uses the entire training set at query time.

Contents

1 Introduction **2**

2 Basics of KNN **2**

3 Other Distance Metrics **4**

4 Applications of KNN **7**

5 Feature Scaling **8**

1 Introduction

K-Nearest Neighbor (KNN) is one of the simplest yet surprisingly effective methods for supervised learning, widely used for classification (and also regression). The key idea is that *the "model" is the data itself*: unlike parametric methods that fit explicit coefficients, KNN stores the entire training set and does not perform parameter learning upfront (a *lazy learner*). For a new instance x, KNN searches the training set for the K most similar instances (its neighbors) under a chosen distance metric (e.g., Euclidean, Manhattan, cosine), then summarizes their outputs—majority vote for classification, or an average for regression—to produce a prediction. This makes KNN conceptually straightforward and often strong when decision boundaries are irregular, but it also shifts computation to query time and makes performance sensitive to several practical choices: the definition of similarity (distance), the value of K (bias–variance tradeoff), optional neighbor weighting, and especially *feature scaling*, since distances can be dominated by variables on larger scales. Throughout this posting, we will formalize these ideas: (i) how distances and scaling shape neighborhoods, (ii) how to select K and weight neighbors, (iii) how prediction differs between classification and regression, and (iv) computational and memory considerations that arise because KNN uses the training dataset *directly* to make predictions.

2 Basics of KNN

The intuition of KNN is straightforward: similar data points tend to belong to similar categories. When a new data point arrives, KNN looks for its K nearest neighbors in the training set and assigns the label based on the majority vote (classification) or averages their values (regression). Unlike other machine learning algorithms, KNN is a **lazy learner**. This means that there is no explicit training process that builds a parametric model. Instead, the "model" is essentially the dataset itself — all computations are deferred until a prediction is requested. Hence, KNN is also referred to as:

- *Instance-based learning*: raw training instances are used directly to make predictions.
- *Lazy learning*: there is no model fitting; all the work happens at prediction time.
- *Non-parametric*: KNN makes no assumptions about the functional form of the problem being solved. As such KNN is referred to as a non-parametric ML algorithm.

Algorithmic Steps for Classification

For a classification task, KNN follows the simple procedure below:

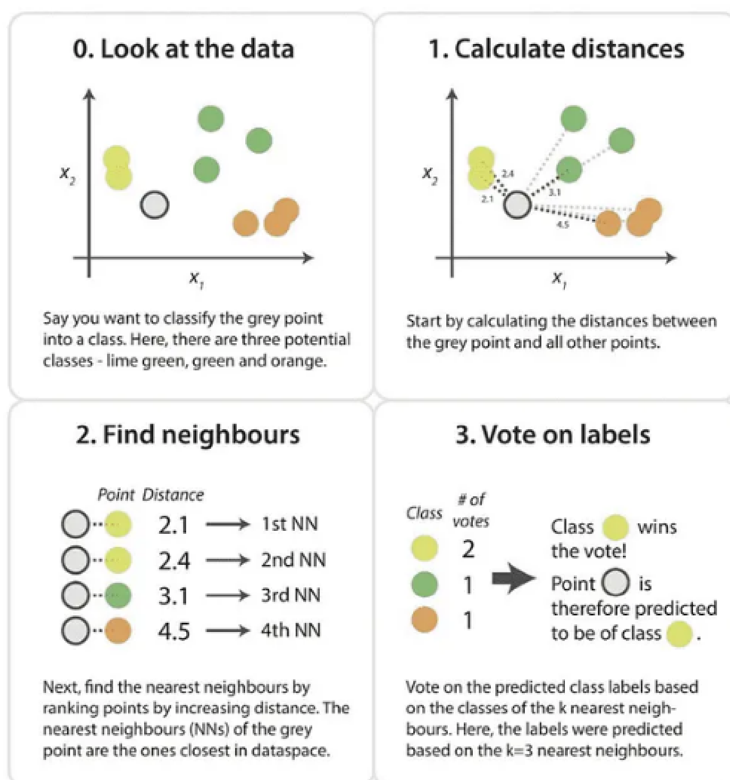
0. Select the number K of neighbors around a new sample that we want to classify or regress (i.e., the number of K is hyper-parameter).

1. Compute the distance between the new sample and every training sample by Euclidean distance (this can be replaced by other distances like Manhattan, cosine, and so on). After that, we need to dentify the K closest neighbors around the new sample.

2. Count how many of these neighbors belong to each category.

3. Assign the new data point to the majority class among the K neighbors.

There is no universal rule for the best K; it is typically chosen by validation. A very small K (e.g., $K = 1$ or 2) makes the model sensitive to noise and outliers, while a very large K oversmooths the decision boundary. (i.e., Small K $\Rightarrow$ low bias, high variance (overfitting), and Large K $\Rightarrow$ high bias, low variance (underfitting).) In practice, $K = 5$ often serves as a reasonable default to start. The example in the figure below is that when K is set to 4, each data has 2 features and 3 possible classes.



Distance Metric

To determine which training samples are most similar to a new input, KNN uses a **distance measure**. For real-valued inputs, the most common choice is the Euclidean distance:

$$d_{\text{Euclidean}}(p, q) = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2},$$

where $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$ are two points in d-dimensional space. This metric captures the straight-line distance implied by the Pythagorean theorem, but other distance measures (e.g., Manhattan, cosine) may be used depending on the data characteristics.

Key Takeaways

The simplicity of KNN makes it a strong baseline for both classification and regression tasks. However, since it stores and scans the entire training data for every query, it can become computationally expensive for large datasets. Moreover, because distances are sensitive to scale, **feature normalization** (e.g., standardization or min–max scaling) is crucial for meaningful similarity comparisons.

3 Other Distance Metrics

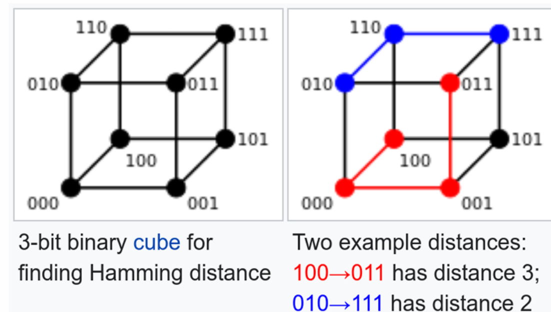
Because distances are sensitive to scale, features are typically standardized (or a metric like Mahalanobis is used when features are correlated). Common alternatives include Manhattan (ℓ_1), Minkowski (p-norm), cosine distance for high-dimensional sparse data, and Hamming distance for binary features.

- **Hamming distance.** In information theory, the Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. In other words, it measures the minimum number of substitutions required to change one string into the other, or the minimum number of errors that could have transformed one string into the other. More formally, for $x, y \in \Sigma^d$,

$$d_H(x, y) = \sum_{j=1}^{d} \mathbf{1}[\, x_j \neq y_j \,]$$

  It is Useful for categorical or binary features (e.g., one-hot vectors).
  Examples: "karolin" vs. "kathrin" = 3; "karolin" vs. "kerstin" = 3; "kathrin" vs "kerstin" = 4; 1011101 vs. 1001001 = 2; 2173896 vs. 2233796 = 3. For binary vectors, it equals the number of bit flips; see the 3-bit cube below (e.g., 100 → 011 has distance 3, and 010 → 111 has distance 2) as seen in the figure below.

- "**karolin**" and "**kathrin**" is 3.
- "**karolin**" and "**kerstin**" is 3.
- "**kathrin**" and "**kerstin**" is 4.
- **1011101** and **1001001** is 2.
- **2173896** and **2233796** is 3.



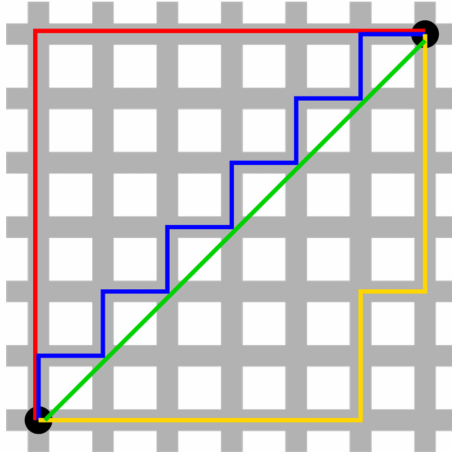
3-bit binary cube for finding Hamming distance

Two example distances: 100→011 has distance 3; 010→111 has distance 2

- **Manhattan distance ($\mathbf{L}_1$).** Also called taxicab geometry, or rectilinear / L1 / snake / city block distance; the distance between two points is the sum of the absolute differences of their Cartesian coordinates. More formally, for $p, q \in \mathbb{R}^d$,

$$d_{\text{Manhattan}}(p, q) = \|p - q\|_1 = \sum_{i=1}^{d} |p_i - q_i|$$

Movement is along axis-aligned streets, so many shortest paths can exist. Look at the example below: from the lower-left to the upper-right corner of the 6×6 grid, the red, yellow, and blue routes all have the same Manhattan length 12, whereas the green line has Euclidean length $6\sqrt{2}$ and is the unique shortest path in Euclidean geometry.



- **Minkowski distance.** The Minkowski distance or Minkowski metric is a metric in a normed vector space which can be considered as a generalization of both the Euclidean distance and the Manhattan distance.

$$d_{\text{Minkowski},\,n}(p,q) = \left(\sum_{i=1}^{d} |p_i - q_i|^n \right)^{1/n}, \qquad n \geq 1,$$

where n is an integer. The above formula is called the Minkowski distance of order n. **Special cases:** $n = 1$ gives Manhattan (L_1), $n = 2$ gives Euclidean (L_2). In the limit $n \to \infty$, we obtain the *Chebyshev (max) distance*:

$$\lim_{n\to\infty} d_{\text{Minkowski},\,n}(p,q) = \max_{1\leq i\leq d} |p_i - q_i|$$

For completeness, the illustration also shows $\lim_{n\to-\infty}(\cdot)^{1/n} = \min_i |p_i - q_i|$.

- **Mahalanobis distance.** The Mahalanobis distance of an observation $\mathbf{x} = (x_1, \ldots, x_d)^\top$ from a set of observations with mean $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_d)^\top$ and covariance matrix $\mathbf{S}$ is:

$$D_{\text{Mahalanobis}}(\mathbf{x}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^\top \mathbf{S}^{-1} (\mathbf{x} - \boldsymbol{\mu})}$$

Mahalanobis distance can also be defined as a dissimilarity measure between two random vectors $\mathbf{x}$ and $\mathbf{y}$ from the same distribution with covariance matrix $\mathbf{S}$:

$$d_{\text{Mahalanobis}}(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})^\top \mathbf{S}^{-1} (\mathbf{x} - \mathbf{y})}$$

If the covariance matrix is the identity matrix ($\mathbf{S} = \mathbf{I}$), the Mahalanobis distance reduces to the Euclidean distance. If the covariance matrix is diagonal($\Sigma = \text{diag}(\sigma_1^2, \ldots, \sigma_d^2)$), then the resulting distance measure is a *standardized Euclidean distance*:

$$d_{\text{Mahalanobis}}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{d} \frac{(x_i - y_i)^2}{s_i^2}},$$

where s_i is the standard deviation of x_i and y_i over the sample set. So each coordinate is a z–score-scaled difference. For $X \sim \mathcal{N}(\mu, \Sigma)$, $D^2 = (X - \mu)^\top \Sigma^{-1}(X - \mu) \sim \chi_d^2$ (useful for outlier detection). Use a well-conditioned $\widehat{\Sigma}$ (e.g., shrinkage or pseudoinverse if singular); optionally choose class-conditional Σ_c in classification to emphasize class-specific geometry.

- **Cosine similarity:** The Cosine distance, also called the cosine similarity, is a measure of similarity between two non-zero vectors of an inner product space. It is defined to equal the cosine of the angle between them, which is also the same as the inner product of the same vectors normalized to both have length 1. The cosine of two non-zero vectors can be derived by using the Euclidean dot product formula such that:

$$\mathbf{p} \cdot \mathbf{q} \;=\; \|\mathbf{p}\|\,\|\mathbf{q}\| \cos\theta,$$

where θ is the angle between the two vectors. Hence,

$$\cos\theta \;=\; \frac{\mathbf{p} \cdot \mathbf{q}}{\|\mathbf{p}\|\,\|\mathbf{q}\|} \;=\; \frac{\sum_{i=1}^{d} p_i q_i}{\sqrt{\sum_{i=1}^{d} p_i^2}\,\sqrt{\sum_{i=1}^{d} q_i^2}}$$

- **Jaccard distance:** The Jaccard index, also called the Jaccard similarity coefficient, is a statistic used for gauging the similarity and diversity of sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

By designing $0 \leq J(A, B) \leq 1$, if both A and B are empty, then $J(A, B) = 1$. The Jaccard distance, which measures dissimilarity between sample sets, is complementary to the Jaccard coefficient and is obtained by subtracting the Jaccard coefficient from 1, or, equivalently, by dividing the difference of the sizes of the union and the intersection of two sets by the size of the union:

$$d_{\text{Jaccard}}(A, B) = 1 - J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

The choice of distance metric should match the data characteristics. For example, Euclidean distance is suitable when features are continuous and comparable, whereas Hamming distance is better for categorical attributes.

4 Applications of KNN

The K-Nearest Neighbors (KNN) is based on a simple idea: points that are close in feature space tend to behave like this. Given a new input, KNN finds the K closest training points and borrows their information to predict. 'K' is obviously a hyper-parameter. You can choose the best distance metric based on the properties of the data. If you are unsure, you can experiment with different distance metrics and different values of K together and see which mix results in the most accurate models. The value of 'K' can be found by algorithm tuning. It is a good idea to try many different values for 'K' (e.g., values from 1 to 21) and see what works best for the problem.

Preparing your data for KNN

Distances drive KNN, so scaling and cleanliness of data matter. These are typical check-points before we use any kind of machine learning, including KNN.

- **Rescale the features.** KNN performs much better when all variables share a comparable scale. Min–max normalization to $[0, 1]$ is a solid baseline; standardization (z-scores) is sensible when features are roughly Gaussian.
- **Handle missing values.** With missing entries, distances are undefined or misleading. Either removing such samples (when safe) or imputing the missing values before running KNN often yield a better performance.
- **Lower the dimensionality.** KNN is strongest in lower dimensions. Feature selection or dimension reduction (e.g., PCA) often improves stability and accuracy on high-d data.

The curse of dimensionality

KNN works well with a small number of input variables, but struggles when the number of inputs is very large. Each extra feature expands the input space and the volume grows exponentially. In high dimensions, even genuinely similar points can appear far apart, and pairwise distances concentrate; the notion of "nearest" becomes noisy. This is the *curse of dimensionality*. The practical remedy is to reduce dimensionality or restrict KNN to informative feature subsets.

How KNN predicts

KNN can be used for both regression and classification even though it is more often to be used for classification. These are the two application modes—regression (aggregate neighbor values) and classification (majority vote)—with K tuned to control stability and sensitivity.

- **KNN for Regression.** Predict by aggregating the responses of the K nearest neighbors: use the *mean* by default, or the *median* for robustness to outliers. Think of K as a noise filter: small K is responsive but unstable; larger K is smoother but may miss local detail. Select K via cross-validation.

- **KNN for Classification.** When KNN is used for classification, the output can be calculated as the class with the highest frequency from the K-most similar instances. Each of the K neighbors votes for its class; the class with the *highest frequency* wins. To reduce ties in binary problems, choose an *odd* K. If a tie occurs, break it consistently (e.g., increase K by one and include the next nearest neighbor). Inversely, use an even number for K when you have an odd number of classes.

Which distance metric should I use?

There is no universal best. Match the metric to your data and validate empirically. However, these are the typical use-cases in the field:

- **Euclidean (L_2).** Strong default when features are on similar scales and represent comparable quantities (e.g., lengths, heights).
- **Manhattan (L_1).** Preferable when coordinate-wise differences matter or when you want less sensitivity to large deviations/outliers.
- **Cosine similarity.** Suited to sparse or directional data (e.g., TF–IDF text vectors), where comparing *angles* is more meaningful than raw coordinate differences.
- **Mahalanobis.** Useful when features are correlated or have very different natural scales; requires a well-conditioned covariance estimate Σ.

In practice, try several metrics and a range of K values (e.g., $K \in \{1, \ldots, 21\}$) with cross-validation, then keep the combination that scores best.

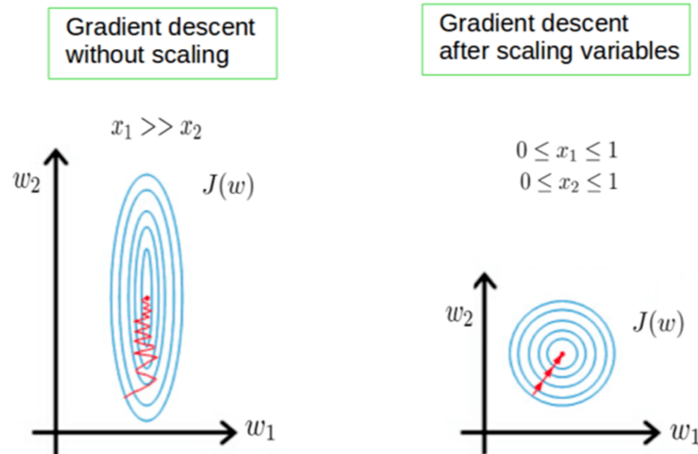
5 Feature Scaling

Feature scaling is a cornerstone of data pre–processing. Distance-based learners like KNN, SVMs (with linear/RBF kernels), and many neural nets implicitly compare *numeric magnitudes*; For example, if one feature spans $[1, 10{,}000]$ while another spans $[1, 10]$, the large–range feature can dominate Euclidean or Manhattan distances and bias predictions. In order to prevent this from blocking machine learning algorithm performance, we need to scale features to be in the same range. Feature scaling is a method that is used to normalize the range of independent variables or features of the data. In data processing, it is also known as data normalization and is generally performed during the data pre-processing step.

Why do we need feature scaling?

- **Proportional contributions.** Ensure each feature contributes roughly proportionally to a distance or dot product, rather than letting wide–range features dominate. For example, the KNN classifier calculates the distance between two points using a distance measure like Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Hence, the range of all features should be normalized so that each feature contributes proportionately to the final distance.

- **Faster optimization.** Gradient descent converges much faster with feature scaling than without it. The difference in ranges of features will cause different step sizes for each feature. To ensure that the gradient descent moves smoothly towards the minima and that the steps for gradient descent are updated at the same rate for all the features, we need to scale the data before feeding it to the model. With comparable scales, gradient descent follows more circular level sets and *converges faster*; otherwise, zig–zagging slows learning.



- **Fair regularization.** Feature scaling is needed if regularization is used as part of the loss function (so that coefficients are penalized appropriately). Penalties such as ℓ_2 or ℓ_1 apply uniformly across features only if they are on compatible scales.
- **Avoid saturation.** In networks with sigmoids/tanh, scaling inputs helps prevent early saturation and vanishing gradients.

When do we need (or not need) scaling?

- **Essential (scale–sensitive):** In general, whenever a learning algorithm is sensitive to the *relative scales of features*—that is, it operates on raw numeric magnitudes rather than ranks—**feature scaling is required** (e.g. KNN, SVMs, linear/logistic regression (especially with regularization), K–means, PCA). Scaling also promotes **faster convergence** in gradient–based methods and helps **avoid saturation** in networks with sigmoids or tanh activations (e.g. most neural networks).
- **Usually unnecessary (scale–invariant rules):** Not every model needs scaling. Algorithms that rely on *rules*—i.e., splits defined by inequalities—are essentially invariant to any *monotonic* rescaling of inputs, so feature scaling has little to no effect. This includes the family of tree–based methods (CART, Random Forests, Gradient–Boosted Trees), where thresholds shift with the scale but the induced ordering and decisions remain the same. Likewise, *generative/linear* models such as Linear Discriminant Analysis (LDA) and Naive Bayes typically account for feature scale through their class–conditional parameters, so normalization rarely changes results materially (though light standardization for numerical stability is still reasonable in practice).

Feature scaling methods

Min–max scaling (normalization). It is the simplest method and consists in rescaling the range of features to scale a fixed range such as in $[0, 1]$ or $[-1, 1]$. The selection of the target range depends on the nature of the data (commonly $[0, 1]$):

$$x_{\text{new}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

The min-max scaler responds well if the standard deviation is small and when a distribution is not Gaussian. It is sensitive to outliers. To map to an arbitrary interval $[a, b]$:

$$x_{\text{new}} = a + (b - a) \times \frac{x - \min(x)}{\max(x) - \min(x)}$$

Mean normalization. Centers by the mean and rescales by the observed range:

$$x_{\text{new}} = \frac{x - \overline{x}}{\max(x) - \min(x)}$$

Standardization (Z–score Normalization). Centers to zero mean and unit variance:

$$x_{\text{new}} = \frac{x - \overline{x}}{\sigma}$$

Standardization centers on and scales each feature *independently*. When a feature is far from normally distributed, z–score scaling is often not the best choice as the mean and standard deviation are *non-robust*. Some outliers can skew the transform and bias downstream modeling.

Robust scaler (median / IQR). As the name suggests, this scaler is *robust to outliers*. When a feature contains many extreme values, scaling by the mean and standard deviation can be misleading; instead, subtract the median and scale by the interquartile range (IQR):

$$x_{\text{new}} = \frac{x - \text{median}(x)}{Q_{75} - Q_{25}},$$

where Q_{25} and Q_{75} are the 25th and 75th percentiles (so $Q_{75} - Q_{25}$ is the IQR). Because both the centering and scaling use *percentiles*, they are far less influenced by a few huge marginal outliers. *Note:* the outliers themselves are still present after the transform, but their leverage on the scale is greatly reduced.

Max–Abs Scaler. Scale each feature by its *maximum absolute value* so that the largest absolute entry becomes 1 (and the smallest -1 if negatives exist):

$$x_{\text{new}} = \frac{x}{\max |x|}$$

This operation does *not* shift/center the data, so sparsity is preserved (no new nonzeros). On strictly nonnegative features, it behaves much like min–max scaling, and therefore it also **suffers from outliers**—an extreme value sets the scale for the entire feature.

Scaling to unit length (vector normalization). Another option is to scale the components of a feature vector such that the complete vector has length one. This usually means dividing each component by the Euclidean length of the vector:

$$x_{\text{new}} = \frac{x}{\|x\|}$$

Normalization vs. Standardization: which to use?

- **Normalization (min–max)** is useful when the data *does not follow a Gaussian distribution*, when a bounded range is desirable (e.g., image pixels in $[0, 1]$), or for models that do not assume any distributional form (e.g., KNN, many neural networks).
- **Standardization (z–score)** is often helpful when features are approximately Gaussian *(though not strictly required)* and for many linear/regularized models.
- **Outliers.** Min–max uses the global min/max, so a few extremes can compress the rest of the data. Z–score has no bounding range and typically distorts the bulk less than min–max, but it is still sensitive to heavy outliers.
- **No hard rule.** In practice, fit and compare models on raw vs. normalized vs. standardized features and keep what validates best.

Practical tips

- **Fit on training, apply to test.** Estimate scaling parameters (min/max, mean/σ, or median/IQR) *only on the training set*, then transform validation/test with those fixed values to avoid data leakage.
- **Match method to data/model.** Use *robust (median/IQR)* when outliers are present; keep *max–abs* to preserve sparsity (no centering); use *unit–length* scaling when cosine/similarity geometry is intended.
- **Document choices.** Record the scaling method and learned parameters for reproducibility and deployment.

This is a posting that I summarized with study-purpose and is adapted from lecture notes of NE-795 (Scientific Machine Learning), given by Professor Xu Wu, NC State University.