K-Nearest Neighbors (KNN)

A Step-by-Step, Practical Guide

This guide walks you through how KNN works from intuition to implementation, with a worked example, tips for choosing hyperparameters, and ready-to-use code.

What is KNN?

K-Nearest Neighbors is a simple, non-parametric, instance-based learning algorithm. To predict for a new sample, it looks at the *k* closest training points (its neighbors) using a distance metric, then **votes** (classification) or **averages** (regression) their targets. No explicit training step is required: the model is essentially the stored dataset.

When should you use KNN?

- You want a strong baseline with minimal tuning.
- Decision boundary may be irregular and non-linear.
- Small to medium datasets (thousands to low millions) and moderate feature counts.
- You can afford slower prediction time or use accelerated nearest-neighbor search.

KNN: Step-by-Step

Training

- Store the training data *X* and labels/targets *y*.
- Optionally scale features (recommended).

Prediction for a new sample x■

1. Choose **k** (number of neighbors).
2. Choose a distance metric (e.g., Euclidean, Manhattan, Minkowski, cosine).
3. Compute distances from x■ to all training points.
4. Find the k points with the smallest distance.
5. **Classification**: majority vote (optionally weighted by 1/distance).
6. **Regression**: average the targets (optionally distance-weighted).
7. Return the predicted class/target.

Common Distance Metrics

- **Euclidean** (L2): d(x, z) = sqrt(Σ (x■ – z■)^2).
- **Manhattan** (L1): d(x, z) = Σ |x■ – z■|.
- **Minkowski**: d(x, z) = (Σ |x■ – z■|■)^(1/p), with p≥1.
- **Cosine**: 1 – (x·z)/(||x|| ||z||), useful for high-dimensional sparse vectors.

Tip: Scale features (e.g., StandardScaler or MinMax) so that units don't distort distance.

Choosing k

- Use cross-validation to tune k. Odd k helps avoid ties in binary classification.
- Small k → low bias, high variance (risk of overfitting).
- Large k → higher bias, lower variance (can oversmooth).
- Heuristic: start with k ≈ √n and search around it.

Weighted KNN

Instead of equal votes, weight each neighbor by an inverse distance, e.g., $w = 1 / (d + \varepsilon)$. This can improve performance when relevant neighbors are very close.

Complexity & Data Structures

- **Training**: O(1) — just store the data (and maybe scale).
- **Prediction**: O(n·d) per query for naive search (n samples, d features).
- Speedups: KD-Tree / Ball-Tree for moderate d; approximate nearest neighbor (ANN) for high d / large n.

Practical Considerations

- **Categorical features**: encode (one-hot/ordinal); consider Hamming for binary.
- **Missing values**: impute before distance computation.
- **Class imbalance**: use class weights, stratified CV, or resampling.
- **Curse of dimensionality**: reduce d (PCA, feature selection) or use ANN methods.

Worked Example (Classification)

We have 2D points with classes A/B. Predict the class of x■ = (3, 4) using Euclidean distance and k = 3.

Point	(x, y)	Class	Distance to x0=(3,4)
P1	(2, 3)	A	1.414
P2	(5, 4)	B	2.000
P3	(3, 2)	A	2.000
P4	(7, 6)	B	4.472
P5	(1, 5)	A	2.236
P6	(4, 5)	B	1.414

Nearest 3 neighbors: P1 (A), P6 (B), P2 (B). Majority vote → **Class B**.

Pseudocode

```
KNN-Predict(x0, X, y, k, metric):
    # X: n×d matrix, y: length-n labels/targets
    # metric: function to compute distance between two d-vectors
    D = []
    for i in 1..n:
        d = metric(x0, X[i])
        D.append((d, y[i]))
    sort D by distance ascending
    N = first k elements of D
    if classification:
        return majority_label(N)   # or weighted by 1/(d+ε)
    else:   # regression
        return average_target(N)   # or weighted average
```

Scikit-learn Example (Classification)

```
from sklearn.preprocessing import StandardScaler from sklearn.neighbors import
KNeighborsClassifier from sklearn.pipeline import make_pipeline from
sklearn.model_selection import cross_val_score pipe = make_pipeline(StandardScaler(),
KNeighborsClassifier(n_neighbors=5, weights='distance', metric='minkowski', p=2))
scores = cross_val_score(pipe, X, y, cv=5) print(scores.mean())
```

Pros & Cons

- **Pros**: simple, no training time, handles non-linear boundaries, competitive baseline.
- **Cons**: slow predictions on large n, sensitive to scale & noise, suffers in high dimensions.

Quick Checklist

- Scale features.
- Tune k with cross-validation (start near √n).
- Try weights='distance'.
- Pick metric suited to data; consider dimensionality reduction.
- For speed, use KD/Ball Trees (moderate d) or ANN methods (high d/large n).

You're set! Use this as a reference when building and tuning KNN models.