L03: KNN & K-Means

Deep Dive: Implementation and Evaluation

Methods and Algorithms

MSc Data Science

Spring 2026

Outline

Deep dive: implementation, evaluation, and Python pipelines

The Math Behind Learning from Neighbors

- Overview: we saw the **intuition**. Now we go deeper
- How do we **measure similarity** precisely?
- How do we **implement** these in Python?
- How do we **validate** our results?

XKCD #1838 by Randall Munroe (CC BY-NC 2.5) — Let's look inside the pile of linear algebra

Learning Objectives

By the end of this deep dive, you will be able to:

1. **Implement KNN** step by step and explain every line of code
2. **Implement K-Means** and trace through each iteration
3. **Evaluate cluster quality** using silhouette analysis
4. **Build a complete pipeline** for classification and clustering in Python

---

Focus: implementation, evaluation, and practical pipeline building

Why Is KNN Called a "Lazy Learner"?

Eager Learners

- Logistic regression, neural networks
- Build a model **during training**
- Training is slow, prediction is fast

KNN — Lazy Learner

- Stores **ALL** training data
- Does all computation at **prediction time**
- Training is instant, prediction is slow

"Lazy because it postpones everything to the last moment."

No parameters to estimate — the training data IS the model

The KNN Algorithm Step by Step

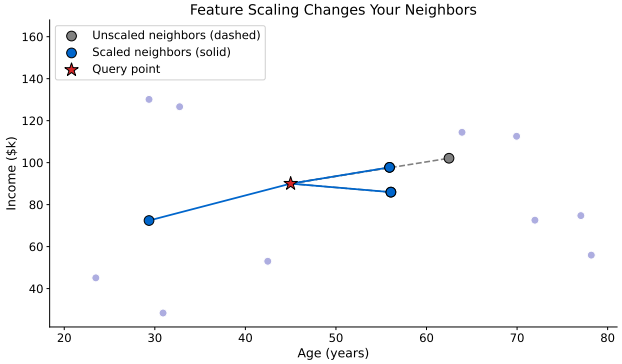
Plain-English pseudocode:

1. **Store** all training data (that's the entire "training" phase)
2. For a new query point: **compute distance** to every stored point
3. **Sort** by distance, pick the K=5 closest neighbors
4. **Count votes** per class among those neighbors
5. **Tie?** Weight by 1/distance — closer neighbors get more say
6. **Return** the majority class

Worked example: 5 neighbors found — 3 vote Class A, 2 vote Class B → **Predict A**

Complexity: $O(n \times p)$ **per prediction — must scan ALL data every time**

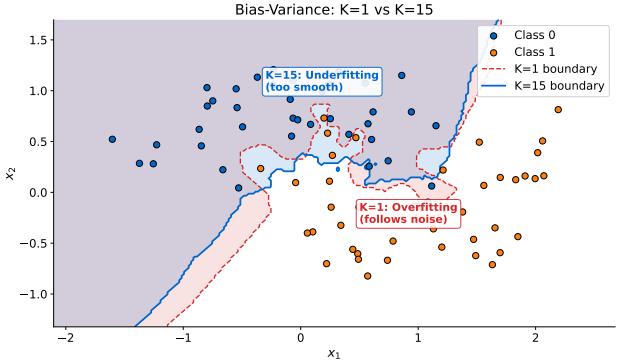
Why Feature Scaling Is Non-Negotiable



- **Before scaling:** income (20k–200k) dominates age (20–80) — KNN picks neighbors based almost entirely on income
- **After scaling:** both features contribute equally — neighbors are truly "similar"
- **Example:** Without scaling, $d = 180{,}000$. After StandardScaler: $d = 2.1$

StandardScaler: subtract mean, divide by standard deviation — do this BEFORE fitting KNN

What Happens as K Changes? Bias vs Variance



- K=1 neighbors: perfectly fits training data but memorizes noise (**overfitting**)
- K=15 neighbors: very smooth boundary, may miss real patterns (**underfitting**)
- Best K balances these — find it with cross-validation

"K=1 is asking ONE friend; K=100 is polling the whole school."

Start with $K=\sqrt{n}$, then tune with cross-validation on held-out data

Weighted KNN: Give Closer Neighbors More Say

Problem: All K neighbors vote equally, even a distant one.

Solution: Weight each neighbor by 1/distance:

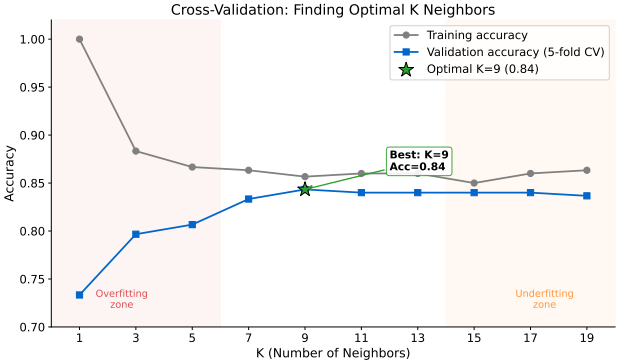
$$\hat{y} = \text{class with highest } \sum_{i=1}^{K} w_i, \quad w_i = \frac{1}{d(\mathbf{x}, \mathbf{x}_i)}$$

Worked example:
3 neighbors at distances 1, 2, 10:

- **Uniform:** each gets weight 1
- **Distance-weighted:**
  weights = 1.0, 0.5, 0.1
  $\rightarrow$ closest neighbor dominates

In scikit-learn: `weights='distance'` **instead of** `'uniform'` **— often improves performance**

How Do We Pick the Best K? Cross-Validation



- Try K neighbors $= 1, 3, 5, \ldots, 19$. For each: split data into 5 folds, average accuracy
- Training accuracy (gray) decreases; validation accuracy (blue) peaks then drops
- **Optimal:** where validation peaks (green star) — here K=5 neighbors

K	Val. Acc.
1	82%
3	87%
5	**89%**
7	88%
9	85%

Cross-validation prevents overfitting to one particular train-test split

When Does KNN Struggle? The Curse of Dimensionality

In **high dimensions** (many features), distances become meaningless — all points are approximately equidistant.

Analogy: *"In a 1D line, nearby houses are easy to find. In a 1000-dimension space, 'nearby' loses meaning because there are too many directions."*

Solutions:

- Reduce dimensions first (PCA — see L05)
- Select only the most relevant features
- Use domain knowledge to drop irrelevant columns

Rule of thumb: KNN works best with fewer than 15–20 features.

If KNN performs poorly, suspect too many features before blaming the algorithm

What Is K-Means Really Optimizing?

Objective: Minimize total distance from each point to its centroid (cluster center):

$$J = \sum_{k=1}^{K} \sum_{\mathbf{x}_i \in C_k} \|\mathbf{x}_i - \boldsymbol{\mu}_k\|^2$$

C_k = set of points in cluster k
$\boldsymbol{\mu}_k$ = centroid of cluster k
J = WCSS (Within-Cluster Sum of Squares)

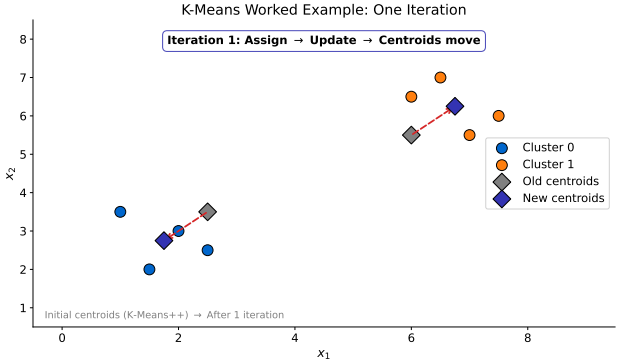
Worked example: 6 points, K=2 clusters
Cluster A: $(1,1), (2,2), (1,3)$
$\rightarrow$ centroid $(1.33, 2.0)$
$\rightarrow$ $\text{WCSS}_A = 1.33$

Cluster B: $(5,5), (6,6), (5,7)$
$\rightarrow$ centroid $(5.33, 6.0)$
$\rightarrow$ $\text{WCSS}_B = 1.33$

Total $J = 2.67$

WCSS measures how tightly packed points are within each cluster — lower is better

K-Means Algorithm: A Worked Visual Example



- Diamonds = initial centroids placed by K-Means++; colored circles = assigned data points
- Dashed arrows show centroids **moving toward** cluster centers after one iteration
- After 3–4 iterations: centroids stabilize, assignments stop changing — **converged!**

Guarantee: WCSS can only decrease at each step → algorithm must stop.

K-Means always converges — but may find a local optimum, not the global best

K-Means++: Why Smart Starting Matters

Problem: Random starting centroids can lead to poor clusters.

K-Means++ solution:
Pick each new centroid **far from** existing ones.

This spreads out the initial centroids, giving the algorithm a much better starting point.

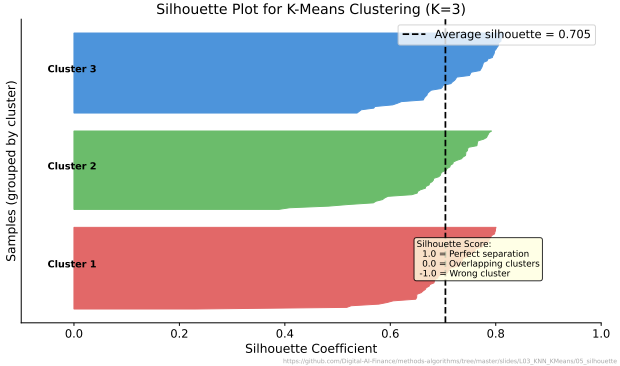
Worked example: $K=3$ clusters needed

1. Pick point $(2, 3)$ randomly
2. Farthest point is $(9, 8) \rightarrow$ pick it
3. Farthest from both is $(2, 9) \rightarrow$ pick it

$\rightarrow$ Centroids are **spread out**, not clumped on the same street!

K-Means++ is the default in scikit-learn — no extra code needed

How Do We Measure Cluster Quality? Silhouette Score

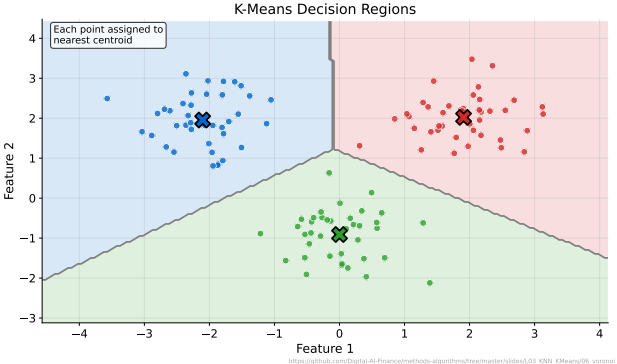


- Each horizontal bar = one data point. Score ranges from -1 to $+1$
- **Near $+1$:** well-clustered. **Near 0:** on boundary. **Below 0:** possibly in wrong cluster
- All clusters should have similar width and stay above the average line

Example: Point at 0.8 $\rightarrow$ clearly right cluster. Point at -0.2 $\rightarrow$ might belong next door.

Compare silhouette plots for K=2, 3, 4, 5 clusters — pick K with highest average silhouette

Why Are K-Means Clusters Always "Round"?



- K-Means divides space into **Voronoi regions** (polygons) — each region contains points closest to one centroid
- These regions are always **convex** (no C-shapes, no donuts)
- K-Means **cannot** discover elongated, ring-shaped, or irregular clusters

If your data has non-round clusters, use DBSCAN or hierarchical clustering instead

What If Clusters Aren't Round? Alternatives

DBSCAN

- Density-based: finds **arbitrary shapes**
- Handles outliers naturally
- No need to specify K clusters

Hierarchical

- Bottom-up merging of points
- Produces a dendrogram (tree diagram)
- Cut at desired level for clusters

Gaussian Mixtures

- Soft assignments (probabilities, not hard labels)
- Handles elliptical clusters
- More flexible than K-Means

K-Means is the starting point — switch to alternatives when validation metrics are poor

K-Means Variants for Special Cases

Mini-Batch K-Means

- Uses random subsets for faster centroid updates
- 10–100$\times$ faster for large data
- Slightly less precise than standard K-Means

K-Medoids (PAM)

- Centroids must be **actual data points** (medoids)
- More robust to outliers
- Works with any distance metric

Mini-Batch for speed on large datasets; K-Medoids for robustness to outliers

KNN in Python: Complete Example

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

# Build pipeline: scale first, then classify
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier(n_neighbors=5,
                                 weights='distance'))
])
pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
print(classification_report(y_test, y_pred))
```

Key params: `n_neighbors`, `weights`, `metric`

Always put StandardScaler in the pipeline — never scale outside cross-validation

K-Means in Python: Complete Example

```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

# Scale features
X_scaled = StandardScaler().fit_transform(X)

# Cluster with K-Means++
kmeans = KMeans(n_clusters=3, init='k-means++',
                n_init=10, random_state=42)
labels = kmeans.fit_predict(X_scaled)
centers = kmeans.cluster_centers_

print(f"Cluster sizes: {np.bincount(labels)}")
print(f"Inertia (WCSS): {kmeans.inertia_:.1f}")
```

Key params: `n_clusters`, `init`, `n_init`

`n_init=10` **means 10 restarts with different seeds — keeps the best result**

Building a Complete Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier())
])
param_grid = {
    'knn__n_neighbors': [3, 5, 7, 11],
    'knn__weights': ['uniform', 'distance']
}
grid = GridSearchCV(pipe, param_grid,
                    cv=5, scoring='f1')
grid.fit(X_train, y_train)
print(f"Best params: {grid.best_params_}")
print(f"Best F1: {grid.best_score_:.3f}")
```

GridSearchCV automates K selection — no manual trial and error needed

Evaluating K-Means: Elbow + Silhouette in Code

```
from sklearn.metrics import silhouette_score

scores, inertias = [], []
K_range = range(2, 9)
for k in K_range:
    km = KMeans(n_clusters=k, n_init=10,
                random_state=42)
    labels = km.fit_predict(X_scaled)
    scores.append(silhouette_score(X_scaled, labels))
    inertias.append(km.inertia_)

best_k = K_range[np.argmax(scores)]
print(f"Best K clusters: {best_k}")
print(f"Silhouette: {max(scores):.3f}")
```

Plot both curves; pick K clusters where both agree.

Combine elbow and silhouette — if they agree, you have strong evidence for K clusters

Hands-On Exercise

1. **Implement weighted KNN:** Compare uniform vs. distance-weighted voting on a fraud dataset — does weighting improve recall?
2. **Run K-Means on customer data:** Interpret cluster profiles using RFM features (Recency, Frequency, Monetary value)
3. **Compare elbow and silhouette** for K cluster selection — do they agree on the best number of clusters?

All exercises use real-world-inspired financial datasets

Key Takeaways

KNN

- Lazy learner — no training phase
- **Scale features** (StandardScaler)
- Choose K neighbors with cross-validation
- Beware high dimensions (>15 features)

K-Means

- Iterative assign-update loop
- Guaranteed convergence (to local optimum)
- Validate with silhouette analysis
- Spherical clusters only

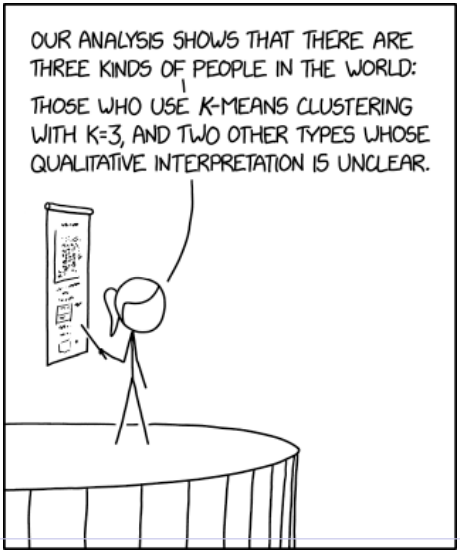
Both: Feature scaling is mandatory. "K" means different things in each algorithm!

Both are foundational — understand them before moving to forests and ensembles

Until Next Time...

"Now you know why 'just cluster it' is never that simple."

Next: L04 — Random Forests: from distance to trees.



XKCD #2731 by Randall Munroe (CC BY-NC 2.5) — Next lecture: ensemble methods

Appendix

Extra Topics and Details

Reference material for further study

Distance Metrics: The Minkowski Family

Generalized Minkowski distance:

$$d_p(\mathbf{x}, \mathbf{x}') = \left(\sum_{j=1}^{p} |x_j - x'_j|^p \right)^{1/p}$$

- p=1: Manhattan (robust to outliers)
- p=2: Euclidean (default)
- p=∞: Chebyshev (max difference)

Worked example:
Points $(1, 3)$ and $(4, 7)$:

Manhattan: $|1-4| + |3-7| = 7$
Euclidean: $\sqrt{9 + 16} = 5$
Chebyshev: $\max(3, 4) = 4$

Higher p amplifies large single-feature differences

Why Does K-Means Always Stop? (Intuition)

Intuitive convergence argument (not a formal proof):

1. Each step can only **improve or maintain** the WCSS score — it never gets worse
2. There are only **finitely many ways** to group n points into K clusters
3. We never revisit a grouping we have already seen $\rightarrow$ **must stop**

Analogy: *"Like rolling a ball downhill — it can only go down or stay put, never roll back up."*

Formal proof uses coordinate descent — see ESL Ch. 14 for details

Computational Complexity: How Fast?

Method	Training	Prediction
KNN (brute force)	$O(1)$	$O(nd)$
KNN (KD-Tree)	$O(nd \log n)$	$O(d \log n)$
K-Means (Lloyd's)	$O(nKdT)$	$O(Kd)$
Mini-Batch K-Means	$O(bKdT)$	$O(Kd)$

n = data points, d = features, K = neighbors or clusters, T = iterations, b = batch size

Note: KD-Tree degrades above 15–20 features (reverts to brute force speed).

For very large data: approximate nearest neighbors (FAISS, Annoy)