K-Nearest Neighbors (KNN) – A Deep Dive

1. What is K-Nearest Neighbors (KNN)?

K-Nearest Neighbors (KNN) is a **simple, yet powerful** supervised learning algorithm used for **classification and regression**.

- **Classification**: Assigns a data point to the **most common class** among its K nearest neighbors.
- **Regression**: Averages the values of K nearest neighbors to predict the output.

How It Works

1. Choose K (the number of nearest neighbors).
2. Compute the **distance** between the new data point and all points in the dataset.
3. Select the **K nearest neighbors**.
4. **For classification**: Assign the most frequent label.
5. **For regression**: Take the average of the K-nearest neighbor values.

2. When to Use & Avoid KNN

✅ **Use KNN When:**

- You have **small to medium-sized datasets**.
- The dataset has **low dimensionality**.
- You need a **non-parametric** model (i.e., you don't assume the data follows a specific distribution).
- You have a **balanced dataset** (classes have similar sample sizes).

❌ **Avoid KNN When:**

- The dataset is **large** → KNN requires computing distances for **every** point at prediction time (slow).
- The dataset has **high dimensionality** → Distance computations become less meaningful (Curse of Dimensionality).
- The dataset is **imbalanced** → Majority class dominates predictions.

3. Choosing K (Number of Neighbors)

Choosing the right **K** is crucial for **accuracy and generalization**.

Effect of K

- **Small K (e.g., K=1,3)** → More variance, sensitive to noise.
- **Large K (e.g., K=20,30)** → More bias, smoother decision boundary.

Methods to Choose K

1. **Cross-Validation**: Try multiple values of K and pick the one with the best validation accuracy.
2. **Elbow Method for Classification**: Plot error rate vs. K and choose the point where error stops decreasing significantly.
3. **Heuristic Rule**: A common rule is:

   $$K = \sqrt{N}$$

   where N is the number of training samples.

---

4. Step-by-Step Pseudo Code for KNN

plaintext Copy Edit

```
# Step 1: Load Dataset
1. Load dataset and split into training and test sets.

# Step 2: Choose K
1. Set the number of neighbors (K).

# Step 3: Compute Distance
1. For each test point:
    a. Compute distance to all training points.
    b. Sort distances in ascending order.

# Step 4: Select K Nearest Neighbors
1. Pick the K closest points.

# Step 5: Make Prediction
1. **For classification**:
    a. Count the occurrences of each class in K neighbors.
    b. Assign the most common class.
2. **For regression**:
    a. Take the mean of K nearest neighbors.

# Step 6: Evaluate Model
1. Compare predictions with true labels.
2. Compute accuracy (classification) or RMSE (regression).
```

5. Mathematical Breakdown

5.1 Distance Calculation

The most common distance metric is **Euclidean Distance**:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Other distance metrics include:

- **Manhattan Distance**:

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

- **Minkowski Distance** (Generalized form of Euclidean & Manhattan):

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

  - $p = 1$ → Manhattan Distance
  - $p = 2$ → Euclidean Distance

5.2 Classification Decision

The predicted class $\hat{y}$ is determined by **majority voting**:

$$\hat{y} = \arg\max_{c} \sum_{i=1}^{K} 1(y_i = c)$$

where:

- c is a class label.
- $1(y_i = c)$ is an indicator function that counts occurrences.

5.3 Regression Decision

For regression, the predicted value is:

$$\hat{y} = \frac{1}{K} \sum_{i=1}^{K} y_i$$

6. Fully Commented Manual Implementation (From Scratch)

python Copy Edit

```python
import numpy as np
from collections import Counter

class KNN:
    """
    Implementation of K-Nearest Neighbors Algorithm (both classification and regres
    """
    def __init__(self, k=3, distance_metric='euclidean'):
        """
        Initialize KNN classifier/regressor.
        :param k: Number of neighbors.
        :param distance_metric: Distance measure ('euclidean' or 'manhattan').
        """
        self.k = k
        self.distance_metric = distance_metric
        self.X_train = None
        self.y_train = None

    def compute_distance(self, x1, x2):
        """
        Compute the distance between two points.
        :param x1: First point.
        :param x2: Second point.
        :return: Distance value.
        """
        if self.distance_metric == 'euclidean':
            return np.sqrt(np.sum((x1 - x2) ** 2))
        elif self.distance_metric == 'manhattan':
            return np.sum(np.abs(x1 - x2))
```

Copy Edit

```
def fit(self, X, y):
    """
    Store training data.
    :param X: Training feature matrix.
    :param y: Training labels.
    """
    self.X_train = X
    self.y_train = y

def predict(self, X, classification=True):
    """
    Predict labels for new data points.
    :param X: Test feature matrix.
    :param classification: If True, performs classification; else regression.
    :return: Predicted labels.
    """
    predictions = []
    for x in X:
        # Compute distances from test point to all training points
        distances = [self.compute_distance(x, x_train) for x_train in self.X_train]
        # Get indices of K closest points
        k_indices = np.argsort(distances)[:self.k]
        # Get K closest labels
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        if classification:
            # Majority vote for classification
            predictions.append(Counter(k_nearest_labels).most_common(1)[0][0])
        else:
            # Mean for regression
            predictions.append(np.mean(k_nearest_labels))

    return np.array(predictions)
```

7. Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Generate synthetic classification dataset
X, y = make_classification(n_samples=300, n_features=5, random_state=42)

# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Initialize and train KNN Classifier
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Predict and evaluate
accuracy = knn.score(X_test, y_test)
print(f"KNN Accuracy: {accuracy:.4f}")
```

Final Takeaways

✅ KNN is a simple yet powerful non-parametric method for classification & regression.
✅ K must be chosen carefully using cross-validation or heuristics.
✅ Sensitive to high-dimensionality and large datasets.

Comparison of Popular Clustering Algorithms

Algorithm	Intuition (How It Works & What It Does)	When to Use	Pros	Cons
K-Means	Assigns points to the nearest centroid, then updates centroids iteratively to minimize variance.	When clusters are **spherical, well-separated**, and you know **K**.	Simple, fast, works well on large datasets.	Sensitive to **initialization** and **outliers**. Struggles with non-spherical clusters.
Hierarchical Clustering	Recursively merges (agglomerative) or splits (divisive) clusters based on similarity.	When you want a **dendrogram** (cluster hierarchy) and **don't know K**.	No need to predefine K, creates a full hierarchy.	Slow for large datasets, sensitive to **distance metrics**.
DBSCAN (Density-Based Clustering)	Groups points that have a minimum number of neighbors within a radius, marking outliers.	When clusters have **arbitrary shapes** and contain **noise**.	Detects **arbitrary-shaped** clusters, robust to noise.	Struggles with varying densities and **high-dimensional data**.
Gaussian Mixture Model (GMM)	Assumes data is generated from **multiple Gaussian distributions** and assigns probabilities to clusters.	When clusters **overlap** and have **elliptical shapes**.	Soft clustering, flexible cluster shapes.	Computationally expensive, requires **Gaussian assumption**.
Mean Shift	Moves data points toward areas of **highest density** (mode seeking).	When clusters have **varying densities** and you want automatic cluster detection.	No need to specify K, works for **irregular densities**.	Computationally expensive, sensitive to bandwidth selection.
Spectral Clustering	Uses graph-based methods and eigenvalues to separate data points into clusters.	When clusters are **connected but not necessarily spherical**.	Works well for **non-convex clusters**, handles complex structures.	Doesn't scale well, computationally expensive.

Final Takeaways

✅ **K-Means** → Best for **simple, well-separated clusters**.

✅ **DBSCAN** → Best for **arbitrary-shaped clusters with noise**.

✅ **Hierarchical Clustering** → Best when **you don't know K and want a hierarchy**.

✅ **GMM** → Best when **clusters overlap and have elliptical shapes**.

✅ **Mean Shift** → Best when **you want automatic cluster detection without K**.

✅ **Spectral Clustering** → Best for **complex structures and non-spherical clusters**.