K-Nearest Neighbors (KNN)

**What is KNN (K-Nearest Neighbor) Algorithm in Machine Learning?**

The k-nearest neighbors (KNN) algorithm is a simple, supervised machine learning method that makes predictions based on how close a data point is to others. It’s widely used for both classification and regression tasks because of its simplicity and popularity.

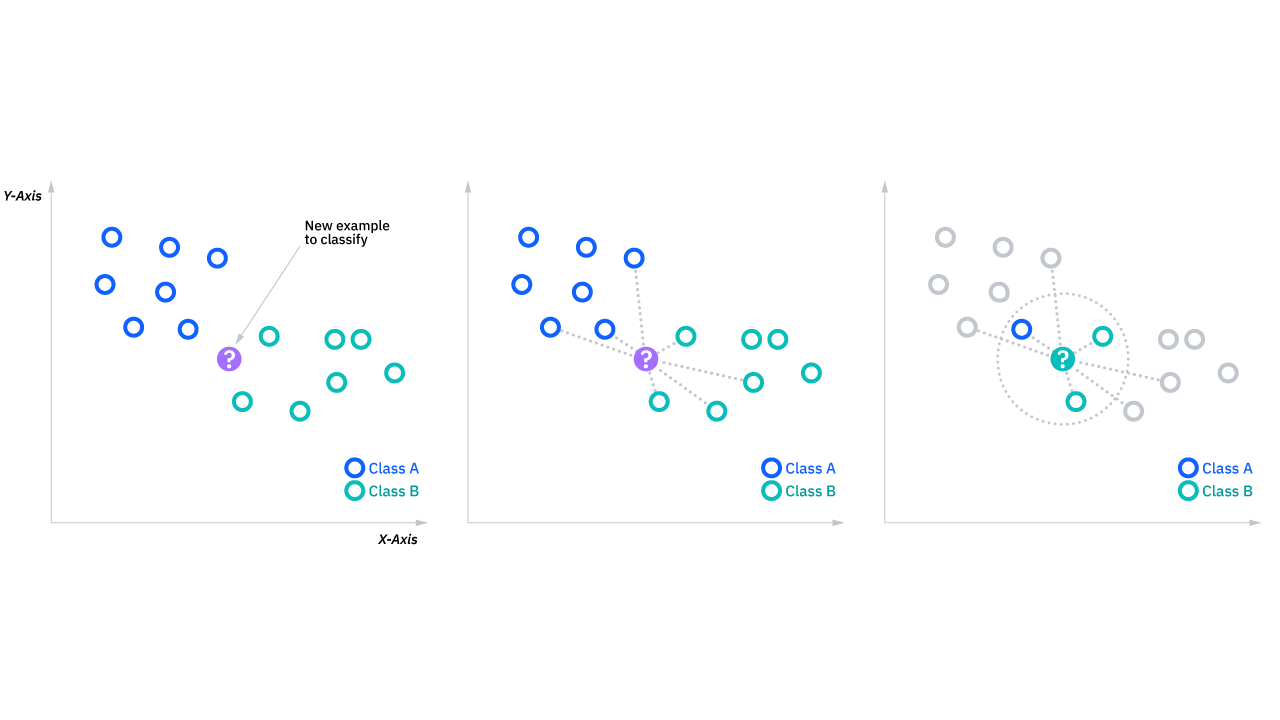
* It classifies a new data point based on the **majority class** of its **K nearest neighbors** in the feature space.
* It is **non-parametric** (makes no assumptions about data distribution).
* It is also called a **lazy learner** since it does not learn a model during training, but stores the entire dataset and makes decisions only when queried.

Next, the algorithm identifies the K nearest neighbors to the input data point based on their distances. In the case of classification, the algorithm assigns the most common class label among the K neighbors as the predicted label for the input data point. For regression, it calculates the average or weighted average of the target values of the K neighbors to predict the value for the input data point.

The KNN algorithm is straightforward and easy to understand, making it a popular choice in various domains. However, its performance can be affected by the choice of K and the distance metric, so careful parameter tuning is necessary for optimal results.

While the KNN algorithm can be used for either regression or classification problems, it is typically used as a classification algorithm, working off the assumption that similar points can be found near one another.

For classification problems, a class label is assigned on the basis of a majority vote—i.e. the label that is most frequently represented around a given data point is used. While this is technically considered “plurality voting”, the term, “majority vote” is more commonly used in literature. The distinction between these terminologies is that “majority voting” technically requires a majority of greater than 50%, which primarily works when there are only two categories. When you have multiple classes—e.g. four categories, you don’t necessarily need 50% of the vote to make a conclusion about a class; you could assign a class label with a vote of greater than 25%. The University of Wisconsin-Madison summarizes this well with an example [here](https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/02_knn_notes.pdf).



KNN diagram

Regression problems use a similar concept as classification problem, but in this case, the average the k nearest neighbors is taken to make a prediction about a classification. The main distinction here is that classification is used for discrete values, whereas regression is used with continuous ones. However, before a classification can be made, the distance must be defined. Euclidean distance is most commonly used, which we’ll delve into more below.

It's also worth noting that the KNN algorithm is also part of a family of “lazy learning” models, meaning that it only stores a training dataset versus undergoing a training stage. This also means that all the computation occurs when a classification or prediction is being made. Since it heavily relies on memory to store all its training data, it is also referred to as an instance-based or memory-based learning method.

Evelyn Fix and Joseph Hodges are credited with the initial ideas around the KNN model in this 1951 [paper](https://apps.dtic.mil/sti/pdfs/ADA800276.pdf) while Thomas Cover expands on their concept in his [research](https://isl.stanford.edu/~cover/papers/transIT/0021cove.pdf), “Nearest Neighbor Pattern Classification.” While it’s not as popular as it once was, it is still one of the first algorithms one learns in data science due to its simplicity and accuracy. However, as a dataset grows, KNN becomes increasingly inefficient, compromising overall model performance. It is commonly used for simple recommendation systems, pattern recognition, data mining, financial market predictions, intrusion detection, and more.

Compute KNN: distance metrics

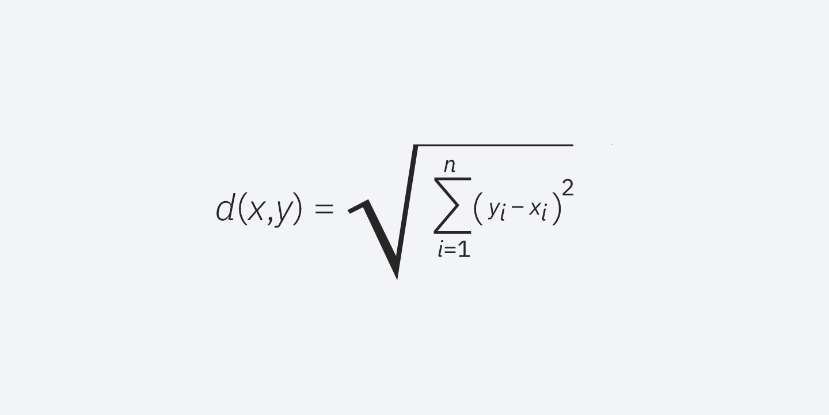
To recap, the goal of the k-nearest neighbor algorithm is to identify the nearest neighbors of a given query point, so that we can assign a class label to that point. In order to do this, KNN has a few requirements:

Determine your distance metrics

In order to determine which data points are closest to a given query point, the distance between the query point and the other data points will need to be calculated. These distance metrics help to form decision boundaries, which partitions query points into different regions. You commonly will see decision boundaries visualized with Voronoi diagrams.

While there are several distance measures that you can choose from, this article will only cover the following:

**Euclidean distance (p=2):** This is the most commonly used distance measure, and it is limited to real-valued vectors. Using the below formula, it measures a straight line between the query point and the other point being measured.



Euclidean distance formula

**✅ When to Use KNN?**

| **Situation** | **Applicability of KNN** |
| --- | --- |
| Small to medium datasets | ✅ Yes, works well |
| High interpretability needed | ✅ Yes, very intuitive |
| Real-time prediction not required | ✅ Yes, training is fast but prediction is slow |
| Low-dimensional data | ✅ Ideal, avoids the curse of dimensionality |

**⚠️ When *Not* to Use KNN?**

* ❌ On **very large datasets**: Prediction becomes **slow**, as it computes distances with all points.
* ❌ On **high-dimensional data**: Distance becomes meaningless (curse of dimensionality).
* ❌ On **noisy datasets**: Sensitive to outliers and irrelevant features.
* ❌ If data is not **normalized**: KNN is distance-based, so feature scaling is important.

**📦 Use Cases of KNN**

**1. Recommendation Systems**

* Suggests products/items based on similar user behavior or product features.
* E.g., “Users who bought this also bought...”

**2. Image Recognition / Classification**

* Classifies images by comparing them to similar labeled images.
* E.g., handwritten digit recognition (like MNIST dataset).

**3. Medical Diagnosis**

* Classifies patients based on similarity to past cases.
* E.g., diagnosing diseases like diabetes, heart conditions, or cancer.

**4. Credit Scoring / Risk Management**

* Categorizes loan applicants based on similar profiles and their repayment histories.

**5. Intrusion Detection**

* Identifies abnormal behavior in network traffic by comparing to past normal/abnormal patterns.

**6. Pattern Recognition**

* Useful in optical character recognition (OCR), speech recognition, and gesture detection.

**📈 Advantages**

* Simple and intuitive.
* No training time.
* Works well with small datasets and well-separated classes.
* Naturally handles multi-class problems.

**🧨 Limitations**

* Slow at prediction time (computationally expensive).
* Requires feature scaling (normalization).
* Sensitive to noise and irrelevant features.
* Struggles with high-dimensional data.

**When Do We Use the KNN Algorithm?**

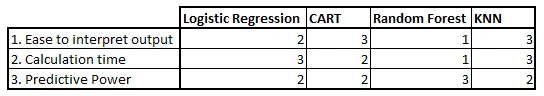
KNN Algorithm can be used for both classification and regression predictive problems. However, it is more widely used in classification problems in the industry. To evaluate any technique, we generally look at 3 important aspects:

1. Ease of interpreting output

2. Calculation time

3. Predictive Power

Let us take a few examples to  place KNN in the scale :



KNN classifier fairs across all parameters of consideration. It is commonly used for its ease of interpretation and low calculation time.

**How Does the KNN Algorithm Work?**

KNN algorithm stores all available cases and classifies new data based on the majority class of its nearest neighbors. Value of **K** in KNN refers to the number of nearest neighbors to consider when performing classification.

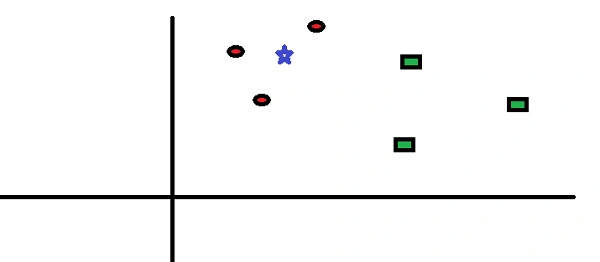
**K** parameter is critical because:

* If **K** is too small, the model may be sensitive to noise in the dataset (The model's prediction can be **easily influenced** by just a **single noisy or incorrect data point**).
* If **K** is too large, the classification might be too generalized (means for each test data you will get same result no differences as you have covered almost all (yes,no) neighbours), and nuances in the data may be overlooked.
* Included neighbors from different classes that are far away.
* Dilute(weaken,reduce) the influence of the actual nearby relevant neighbors.
* Miss important local structure or subgroup behavior.

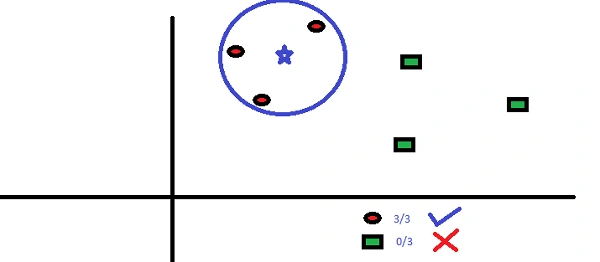
**In machine learning, noise refers to:**

* Incorrect labels (e.g., mislabelled samples)
* Outliers (data points far from the general trend)
* Irrelevant or distorted features
* Random fluctuations that don’t represent true patterns

Let’s take a simple case to understand this algorithm. Following is a spread of red circles (RC) and green squares (GS):



You intend to find out the class of the blue star (BS). BS can either be RC or GS and nothing else. The “K” in KNN algorithm is the nearest neighbor we wish to take the vote from. Let’s say K = 3. Hence, we will now make a circle with BS as the center just as big as to enclose only three data points on the plane. Refer to the following diagram for more details:



The three closest points to BS are all RC. Hence, with a good confidence level, we can say that the BS should belong to the class RC. Here, the choice became obvious as all three votes from the closest neighbor went to RC. The choice of the parameter K is very crucial in this algorithm. Next, we will understand the factors to be considered to conclude the best K.

**How Do We Choose the Factor K?**

**Try Multiple Values of K (Hyperparameter Tuning)**

We test different values of K (like 1, 3, 5, 7, …) and pick the one that gives **best accuracy** on **validation data** using methods like:

* **Cross-validation**
* **GridSearchCV / RandomizedSearchCV**

**2. Odd Values for Classification**

* Use **odd K values** (like 3, 5, 7) to avoid **ties** when doing majority voting in binary classification.

**✅ 3. Rule of Thumb (initial guess)**

A common starting point:

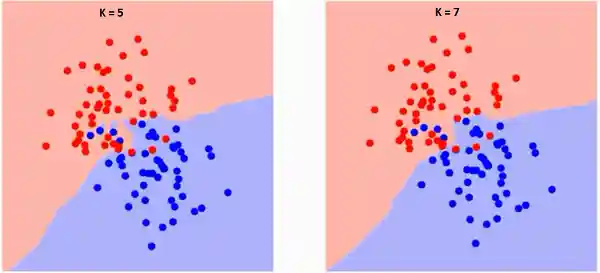


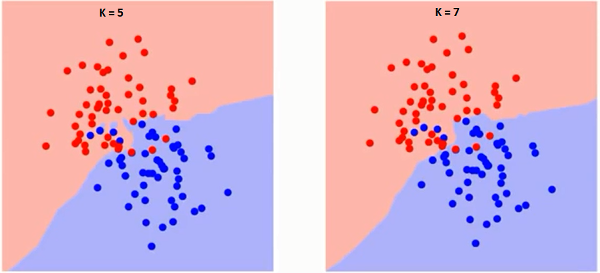
Where **N** is the number of training samples. Then try nearby values to tune.

**✅ 4. Avoid Extremes**

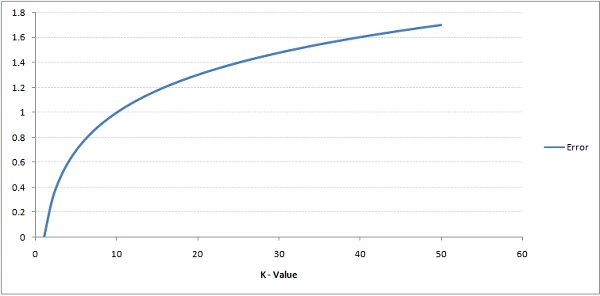
* **Too small (e.g., K=1)** → overfits, too sensitive to noise
* **Too large (e.g., K=100)** → underfits, ignores local structure

First, let us try to understand the influence of the K-nearest neighbors (KNN) in the algorithm. If we consider the last example, keeping all 6 training observations constant, a given K value allows us to establish boundaries for each class. These decision boundaries effectively segregate, for instance, RC from GS. Similarly, let’s examine the impact of the value “K” on these class boundaries. The following illustrates the distinct boundaries that separate the two classes, each corresponding to different values of K.

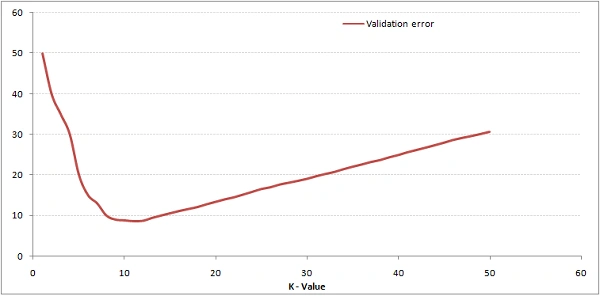




If you watch carefully, you can see that the boundary becomes smoother with increasing value of K. With K increasing to infinity it finally becomes all blue or all red depending on the total majority.  The training error rate and the validation error rate are two parameters we need to access different K-value. Following is the curve for the training error rate with a varying value of K :



As you can see, the error rate at K=1 is always zero for the training sample. This is because the closest point to any training data point is itself.Hence the prediction is always accurate with K=1. If validation error curve would have been similar, our choice of K would have been 1. Following is the validation error curve with varying value of K:



This makes the story more clear. At K=1, we were overfitting the boundaries. Hence, error rate initially decreases and reaches a minima. After the minima point, it then increase with increasing K. To get the optimal value of K, you can segregate the training and validation from the initial dataset. Now plot the validation error curve to get the optimal value of K. This value of K should be used for all predictions.

**What is the decision surface or boundary:**

The **decision boundary** in KNN is the **dividing line (or shape)** that separates different classes based on **how nearby training points vote**. It is **non-linear**, **data-driven**, and changes with **K**.

A **decision boundary** in KNN is an **imaginary line (in 2D)** or **surface (in higher dimensions)** that **separates different classes** based on which class the K-nearest neighbors vote for.

It tells you:  
➡️ *"If a new data point falls on this side of the boundary, classify it as Class A; otherwise, Class B."*

**Example Visualization:**

Imagine 2 classes: red and blue.  
KNN looks at neighbors and draws curvy lines to separate red regions from blue ones — that's the **decision boundary**.

* **Small K (e.g., 1)** → Boundary is very detailed and jagged (overfitting).
* **Large K (e.g., 15)** → Boundary is smoother and more general (less sensitive).

**🧪 Code Example: Visualizing Decision Boundary**

python

CopyEdit

from sklearn.datasets import make\_classification

from sklearn.neighbors import KNeighborsClassifier

import matplotlib.pyplot as plt

from mlxtend.plotting import plot\_decision\_regions

X, y = make\_classification(n\_features=2, n\_redundant=0, n\_informative=2,

n\_clusters\_per\_class=1, n\_samples=100, random\_state=1)

knn = KNeighborsClassifier(n\_neighbors=3)

knn.fit(X, y)

plot\_decision\_regions(X, y, clf=knn)

plt.title('KNN Decision Boundary (k=3)')

plt.show()

**How the decision surface is generated**

A **decision surface** (or boundary) is a **visual representation** showing which areas of the feature space belong to which class.

**🛠️ How It’s Generated in KNN**

KNN doesn't build an equation. Instead, it follows a **brute-force approach**:

**Step-by-Step:**

1. **Create a Grid of Points:**
   * Cover the entire feature space with a fine grid (like pixels in an image).
   * Each point is a fake “test” point.
2. **Use KNN to Predict Each Grid Point:**
   * For each point on the grid, calculate its **K nearest neighbors** from the training data.
   * **Vote** among those neighbors to assign a class.
3. **Color the Grid Point Based on Class:**
   * If the majority neighbors are from class A → color it blue, for example.
   * If majority from class B → color it red.
4. **The Edges Where Color Changes = Decision Boundary**
   * The line (or surface) where the predicted class changes marks the **decision surface**.

**🧪 Example (2D)**

In 2D space, we do this:

python

CopyEdit

import numpy as np

import matplotlib.pyplot as plt

from sklearn.neighbors import KNeighborsClassifier

from sklearn.datasets import make\_classification

# Create sample 2D data

X, y = make\_classification(n\_features=2, n\_informative=2,

n\_redundant=0, n\_clusters\_per\_class=1,

n\_samples=100, random\_state=42)

# Train KNN

knn = KNeighborsClassifier(n\_neighbors=5)

knn.fit(X, y)

# Step 1: Create a mesh grid

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 200),

np.linspace(y\_min, y\_max, 200))

# Step 2: Predict each point on the grid

Z = knn.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

# Step 3: Plot decision surface

plt.contourf(xx, yy, Z, alpha=0.3)

plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')

plt.title("KNN Decision Surface (k=5)")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

**Generate numpy mushgrid**

numpy.meshgrid() is a function used to **create coordinate grids** from two 1D arrays. It's especially useful for plotting 2D functions and decision boundaries (like in KNN).

**🔍 Simple Definition:**

meshgrid turns two 1D arrays of **x** and **y** values into **2D grid matrices**, so that every combination of (x, y) can be evaluated.

**✅ Use Case Example:**

python

CopyEdit

import numpy as np

x = np.array([1, 2, 3]) # x-coordinates

y = np.array([4, 5]) # y-coordinates

X, Y = np.meshgrid(x, y)

print("X:\n", X)

print("Y:\n", Y)

**🧾 Output:**

lua

CopyEdit

X:

[[1 2 3]

[1 2 3]]

Y:

[[4 4 4]

[5 5 5]]

This creates a 2D grid of coordinates:

| **(1,4)** | **(2,4)** | **(3,4)** |
| --- | --- | --- |
| (1,5) | (2,5) | (3,5) |

**🎨 Visual Explanation:**

X gives all the x-values in the grid.  
Y gives all the y-values in the grid.  
You can think of these as pixel coordinates for an image or plot.

**🧠 Common Use in ML:**

In KNN or SVM, we use meshgrid to generate all the test points across the plot space so we can color them and **visualize the decision boundary**.

**Mlxtends for decision surface**

**Mlxtend** (short for **Machine Learning Extensions**) is an open-source Python library by **Sebastian Raschka** that provides helpful tools for machine learning and data science workflows.

You can install it using:

bash

CopyEdit

pip install mlxtend

**📌 Key Module for Decision Surface**

The most relevant part for decision boundaries is:

python

CopyEdit

from mlxtend.plotting import plot\_decision\_regions

This function helps **easily visualize decision surfaces** for classifiers in 2D (i.e., two feature dimensions).

**🔍 Why Use plot\_decision\_regions?**

* ✅ Easy to use — one function call
* ✅ Works with most **Scikit-learn-compatible classifiers**
* ✅ Supports **binary and multiclass classification**
* ✅ Automatically plots the decision surface **based on model predictions**

**🧪 Example with KNN**

python

CopyEdit

from sklearn.neighbors import KNeighborsClassifier

from sklearn.datasets import make\_classification

from mlxtend.plotting import plot\_decision\_regions

import matplotlib.pyplot as plt

# Step 1: Create 2D dataset

X, y = make\_classification(n\_samples=100, n\_features=2,

n\_redundant=0, n\_informative=2,

n\_clusters\_per\_class=1, random\_state=1)

# Step 2: Fit a KNN model

knn = KNeighborsClassifier(n\_neighbors=5)

knn.fit(X, y)

# Step 3: Plot decision surface

plot\_decision\_regions(X, y, clf=knn)

plt.title("KNN Decision Surface (k=5)")

plt.show()

**🛠️ How It Works Internally**

Under the hood, plot\_decision\_regions:

1. Creates a **mesh grid** over the feature space (using numpy.meshgrid)
2. Uses the classifier’s .predict() method on each point in the grid
3. Colors each grid point based on predicted class
4. Plots the training data points on top

**⚠️ Limitations**

* Only supports **2D data (2 features)**. If your dataset has more than two features, you'll need to reduce it using:
  + Feature selection (manually pick two)
  + Dimensionality reduction (e.g., PCA)
* Doesn’t work with regressors (classification only)

**Overfitting and underfitting in KNN**

Overfitting Occurs when K is too small, especially K = 1

Underfitting in KNN Occurs when K is too large (e.g., K = 100)

**Limitations of KNN**

1. **Computationally Expensive at Prediction Time**
   * KNN **stores all training data** and compares the test point with every training example.
   * Prediction involves calculating distances to all training points → slow with large datasets.
2. **Curse of Dimensionality**
   * In high-dimensional spaces, distance metrics become less meaningful.
   * Points tend to become equidistant → reduces KNN’s effectiveness.
   * Requires dimensionality reduction or feature selection.
3. **Sensitive to Irrelevant Features and Noise**
   * KNN uses **all features equally** when computing distances.
   * Irrelevant or noisy features can degrade performance.
   * Needs careful feature engineering or feature scaling.
4. **Choosing the Right K is Critical**
   * Small K → sensitive to noise (overfitting).
   * Large K → may oversmooth and miss patterns (underfitting).
   * Requires hyperparameter tuning.
5. **Imbalanced Data Problems**
   * If one class dominates, neighbors are likely from that class → bias.
   * Needs techniques like weighted voting or data balancing.
6. **Memory Intensive**
   * Since it stores the entire dataset, it requires significant memory for large datasets.

**💤 Why is KNN called a Lazy Learning Algorithm?**

* **Lazy Learning:** defers the model building until a query (prediction) is made.
* KNN **does NOT train a model** upfront.
* It **just stores the training data** as is.
* When a new instance comes in, KNN performs computation **on-the-fly** (distance calculations and voting).
* No explicit generalization or parameter estimation happens before prediction.

This contrasts with **eager learners** (like logistic regression, decision trees) which build a generalized model during training.

**KNN with High-Dimensional Data (Curse of Dimensionality)**

* **Problem:**
  + As the number of features (dimensions) increases, **all points tend to look equally distant** because distances get diluted.
  + KNN relies on distance metrics (e.g., Euclidean), but in high dimensions, distance loses meaning.
  + Leads to poor discrimination between neighbors → bad predictions.
* **Why it happens:**
  + Volume of space grows exponentially.
  + Data becomes sparse; neighbors are not “close” anymore.
* **Remedies:**
  + **Dimensionality Reduction**: PCA, t-SNE, UMAP to reduce features to 2-3 key components.
  + **Feature Selection**: Select only the most important features.
  + Use distance metrics more robust to high dimensions (e.g., cosine similarity).

**2. KNN with Outliers**

* **Problem:**
  + Outliers are rare points far from others.
  + KNN is sensitive because it considers the nearest neighbors without differentiating normal points from outliers.
  + Outliers can wrongly influence the classification of nearby points.
* **Remedies:**
  + **Outlier Detection & Removal** before applying KNN.
  + Use **robust distance metrics** or weighting neighbors by distance (closer neighbors weigh more).
  + Increase K to smooth out the effect of outliers.

**3. KNN with Non-Homogeneous Scales (Different Feature Scales)**

* **Problem:**
  + Features with larger numeric ranges dominate distance calculations.
  + Example: Age in 0–100 vs. Income in 0–100,000 → Income will dominate Euclidean distance.
* **Remedies:**
  + **Feature Scaling / Normalization**: StandardScaler (z-score), MinMaxScaler to bring all features to a similar scale.
  + Apply scaling consistently on train and test data.

**4. KNN with Imbalanced Data**

* **Problem:**
  + If one class is much more frequent, the majority class will dominate neighbors.
  + KNN will tend to predict the majority class, ignoring minority classes.
* **Remedies:**
  + Use **weighted voting** where closer neighbors have more influence.
  + Use **resampling techniques**:
    - Oversampling minority class (SMOTE, ADASYN)
    - Undersampling majority class
  + Use **distance-weighted KNN** (inverse distance weighting).

**Summary Table**

| **Challenge** | **Why it affects KNN** | **Typical Fixes** |
| --- | --- | --- |
| High dimensionality | Distances lose meaning | Dimensionality reduction, feature selection |
| Outliers | Distorts neighbor selection | Outlier removal, distance weighting, increase K |
| Different scales | Features with large scales dominate | Feature scaling (normalization/standardization) |
| Imbalanced data | Majority class dominates votes | Weighted voting, resampling (oversampling/undersampling) |

**Goal: Classify iris flower species using KNN**

**✅ Step-by-Step KNN Implementation (Iris Dataset Example)**

python

CopyEdit

# Step 1: Import Libraries and Load Data

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from collections import Counter

# Load dataset

iris = load\_iris()

X, y = iris.data, iris.target

python

CopyEdit

# Step 2: Normalize Data (Feature Scaling)

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

python

CopyEdit

# Step 3: Split Data – Train/Test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

python

CopyEdit

# Step 4: Implement KNN (from Scratch)

class KNNClassifier:

def \_\_init\_\_(self, k=3):

self.k = k

def fit(self, X, y):

self.X\_train = X

self.y\_train = y

def \_euclidean\_distance(self, x1, x2):

return np.sqrt(np.sum((x1 - x2) \*\* 2))

def \_predict\_single(self, x):

distances = [self.\_euclidean\_distance(x, x\_train) for x\_train in self.X\_train]

k\_indices = np.argsort(distances)[:self.k]

k\_nearest\_labels = [self.y\_train[i] for i in k\_indices]

most\_common = Counter(k\_nearest\_labels).most\_common(1)

return most\_common[0][0]

def predict(self, X):

return np.array([self.\_predict\_single(x) for x in X])

python

CopyEdit

# Step 5: Train Model and Apply on One Test Sample

knn = KNNClassifier(k=3)

knn.fit(X\_train, y\_train)

# Predict one test sample

sample\_index = 0

sample = X\_test[sample\_index].reshape(1, -1)

predicted\_label = knn.predict(sample)[0]

print(f"Predicted: {iris.target\_names[predicted\_label]}, Actual: {iris.target\_names[y\_test[sample\_index]]}")

python

CopyEdit

# Step 6: Calculate Accuracy on Full Test Set

y\_pred = knn.predict(X\_test)

accuracy = np.mean(y\_pred == y\_test)

print(f"Accuracy on test set: {accuracy \* 100:.2f}%")

**✅ Sample Output**

vbnet

CopyEdit

Predicted: versicolor, Actual: versicolor

Accuracy on test set: 96.67%

**📝 Summary**

* ✔️ You learned how to apply KNN on a real-world dataset.
* ✔️ Followed a clean pipeline: import, scaling, split, fit, predict, evaluate.
* ✔️ Achieved high accuracy using 3-NN on normalized Iris data.

Would you like to try the same with a multi-class confusion matrix or plot decision boundaries?

4o

**You said:**

please give me one more code example for multidimentional data data and use regrassion

**ChatGPT said:**

Great! Let's walk through a **KNN Regression** example using **multidimensional data**. We'll use the **California Housing dataset**, which is a common regression benchmark.

**🎯 Goal: Predict housing prices using KNN Regression (from scratch)**

We'll follow the same structure:

**✅ Steps:**

1. Import Libraries and Load Data
2. Normalize Data
3. Split into Train/Test
4. Implement KNN for Regression
5. Predict a Single Sample
6. Evaluate with MSE / R² Score

**🧩 Step-by-Step Code**

python

CopyEdit

# Step 1: Import Libraries and Load Data

import numpy as np

from sklearn.datasets import fetch\_california\_housing

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error, r2\_score

python

CopyEdit

# Load dataset

housing = fetch\_california\_housing()

X, y = housing.data, housing.target # X shape: (20640, 8), y shape: (20640,)

python

CopyEdit

# Step 2: Normalize Data (Feature Scaling)

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

python

CopyEdit

# Step 3: Train/Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

python

CopyEdit

# Step 4: KNN Regressor from Scratch

class KNNRegressor:

def \_\_init\_\_(self, k=5):

self.k = k

def fit(self, X, y):

self.X\_train = X

self.y\_train = y

def \_euclidean\_distance(self, x1, x2):

return np.sqrt(np.sum((x1 - x2) \*\* 2))

def \_predict\_single(self, x):

distances = [self.\_euclidean\_distance(x, point) for point in self.X\_train]

k\_indices = np.argsort(distances)[:self.k]

k\_nearest\_values = [self.y\_train[i] for i in k\_indices]

return np.mean(k\_nearest\_values)

def predict(self, X):

return np.array([self.\_predict\_single(x) for x in X])

python

CopyEdit

# Step 5: Train Model and Predict One Sample

knn\_reg = KNNRegressor(k=5)

knn\_reg.fit(X\_train, y\_train)

# Predict on one test sample

sample\_idx = 0

sample = X\_test[sample\_idx].reshape(1, -1)

predicted\_value = knn\_reg.predict(sample)[0]

print(f"Predicted Price: {predicted\_value:.2f}, Actual Price: {y\_test[sample\_idx]:.2f}")

python

CopyEdit

# Step 6: Evaluate on Full Test Set

y\_pred = knn\_reg.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"Mean Squared Error (MSE): {mse:.2f}")

print(f"R² Score: {r2:.2f}")

**✅ Example Output**

yaml

CopyEdit

Predicted Price: 2.45, Actual Price: 2.33

Mean Squared Error (MSE): 0.53

R² Score: 0.72

**📌 Notes:**

* The dataset has 8 features (multidimensional).
* KNN for regression takes the **mean** of the k nearest target values.
* R² Score close to 1 is good. MSE close to 0 is ideal.

Would you like me to show how this compares to sklearn’s built-in KNeighborsRegressor or plot prediction vs actual?

**Lenear Regression Example:**

two **realistic regression examples** that predict continuous outputs using:

1. ✅ **Simple Linear Regression** – One independent variable
2. ✅ **Multiple Linear Regression** – Multiple independent variables

We'll use **scikit-learn** and real datasets to stay practical.

**1. Simple Linear Regression: Predicting Wine Quality Based on Alcohol Content**

**Dataset:** [Wine Quality Dataset](https://archive.ics.uci.edu/ml/datasets/wine+quality)

**Objective:** Predict the quality of red wine based on its alcohol content.

**🔧 Steps:**

1. **Import Libraries and Load Data**

python

CopyEdit

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score

# Load dataset

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'

data = pd.read\_csv(url, sep=';')

1. **Select Features and Target**

python

CopyEdit

X = data[['alcohol']] # Independent variable

y = data['quality'] # Dependent variable

1. **Split Data into Training and Testing Sets**

python

CopyEdit

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

1. **Train the Model**

python

CopyEdit

model = LinearRegression()

model.fit(X\_train, y\_train)

1. **Make Predictions and Evaluate**

python

CopyEdit

y\_pred = model.predict(X\_test)

print(f"R² Score: {r2\_score(y\_test, y\_pred):.2f}")

print(f"MSE: {mean\_squared\_error(y\_test, y\_pred):.2f}")

1. **Visualize the Results**

python

CopyEdit

plt.scatter(X\_test, y\_test, color='blue', label='Actual')

plt.plot(X\_test, y\_pred, color='red', linewidth=2, label='Predicted')

plt.xlabel('Alcohol Content')

plt.ylabel('Wine Quality')

plt.title('Simple Linear Regression')

plt.legend()

plt.show()

**📈 2. Multiple Linear Regression: Predicting Wine Quality Based on Multiple Features**

**Dataset:** [Wine Quality Dataset](https://archive.ics.uci.edu/ml/datasets/wine+quality)

**Objective:** Predict the quality of red wine based on various physicochemical tests.

**🔧 Steps:**

1. **Import Libraries and Load Data**

python

CopyEdit

import pandas as pd

import numpy as np

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.preprocessing import StandardScaler

# Load dataset

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'

data = pd.read\_csv(url, sep=';')

1. **Select Features and Target**

python

CopyEdit

X = data.drop('quality', axis=1) # Independent variables

y = data['quality'] # Dependent variable

1. **Normalize Features**

python

CopyEdit

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

1. **Split Data into Training and Testing Sets**

python

CopyEdit

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

1. **Train the Model**

python

CopyEdit

model = LinearRegression()

model.fit(X\_train, y\_train)

1. **Make Predictions and Evaluate**

python

CopyEdit

y\_pred = model.predict(X\_test)

print(f"R² Score: {r2\_score(y\_test, y\_pred):.2f}")

print(f"MSE: {mean\_squared\_error(y\_test, y\_pred):.2f}")

**📝 Summary**

* **Simple Linear Regression** considers a single feature (alcohol content) to predict wine quality.
* **Multiple Linear Regression** utilizes multiple features (e.g., acidity, sugar, pH) for a more comprehensive prediction.

Both models demonstrate how regression techniques can be applied to real-world datasets to predict continuous outcomes.

If you have any further questions or need assistance with other datasets, feel free to ask!