**K-Nearest Neighbors (KNN)**

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).

**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.

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.

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

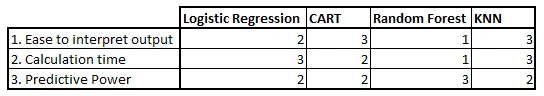
KNN Algorithm can be used for both classification and regression predictive problems. However, it is more widely used in classification problems in the industry. To evaluate any technique, we generally look at 3 important aspects:

1. Ease of interpreting output

2. Calculation time

3. Predictive Power

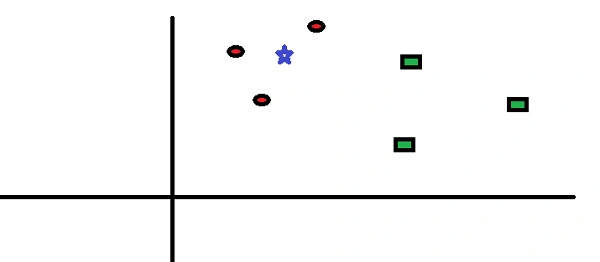
Let us take a few examples to  place KNN in the scale :



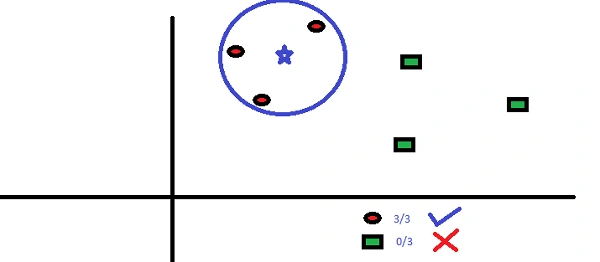
KNN classifier fairs across all parameters of consideration. It is commonly used for its ease of interpretation and low calculation time.

**How Does the KNN Algorithm Work?**

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



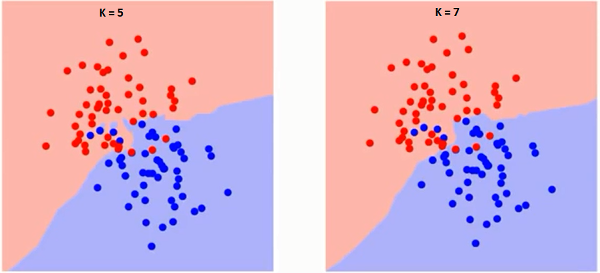
You intend to find out the class of the blue star (BS). BS can either be RC or GS and nothing else. The “K” in KNN algorithm is the nearest neighbor we wish to take the vote from. Let’s say K = 3. Hence, we will now make a circle with BS as the center just as big as to enclose only three data points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



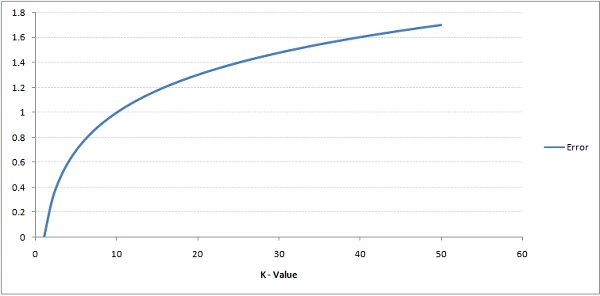
The three closest points to BS are all RC. Hence, with a good confidence level, we can say that the BS should belong to the class RC. Here, the choice became obvious as all three votes from the closest neighbor went to RC. The choice of the parameter K is very crucial in this algorithm. Next, we will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

How Do We Choose the Factor K?

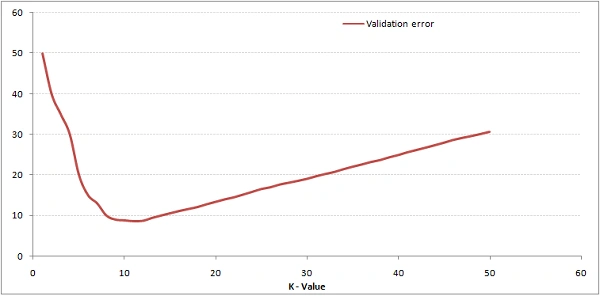
First, let us try to understand the influence of the K-nearest neighbors (KNN) in the algorithm. If we consider the last example, keeping all 6 training observations constant, a given K value allows us to establish boundaries for each class. These decision boundaries effectively segregate, for instance, 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.

***The above content can be understood more intuitively using our free course –***[***K-Nearest Neighbors (KNN) Algorithm in Python and R***](https://courses.analyticsvidhya.com/courses/K-Nearest-Neighbors-KNN-Algorithm?utm_source=blog&utm_medium=knn_in_python&R)

Breaking It Down – Pseudo Code of KNN

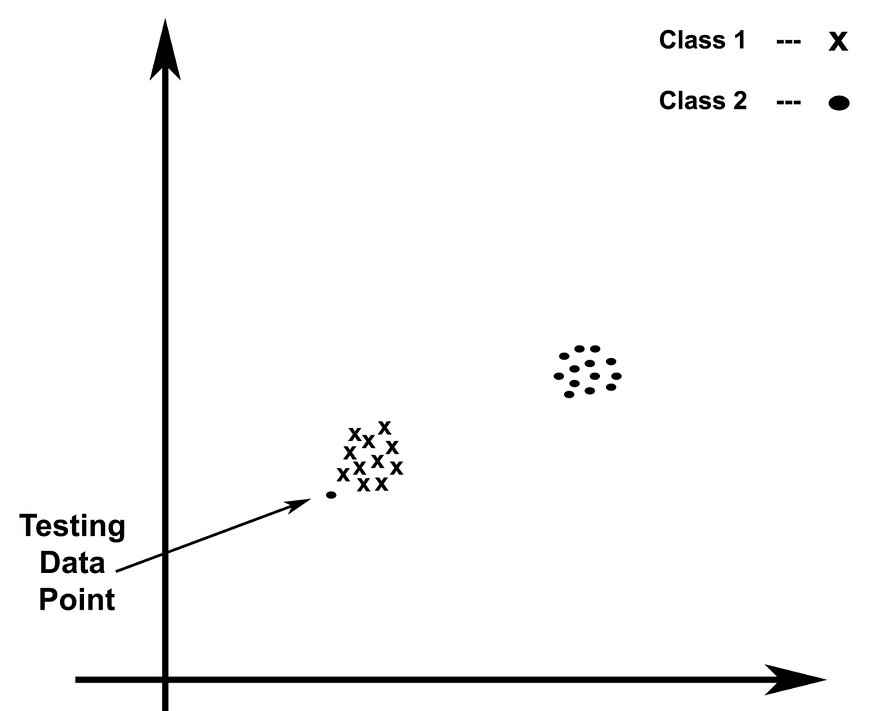
We can implement a KNN model by following the below steps:

1. Load the data
2. Initialise the value of k
3. For getting the predicted class, iterate from 1 to total number of training data points
   * Calculate the distance between test data and each row of training dataset. Here we will use Euclidean distance as our distance metric since it’s the most popular method. The other distance functions or metrics that you can use include Manhattan distance, Minkowski distance, Chebyshev, cosine, etc. If you have categorical variables, you can use Hamming distance.
   * Sort the calculated distances in ascending order based on distance values
   * Get top k rows from the sorted array
   * Get the most frequent class of these rows
   * Return the predicted class

**Choosing the Right Number of Neighbors**

The key to using KNN effectively is selecting the right value for “K,” which is the number of neighbors we’ll consider making predictions. If we choose K = 1, makes our model more prone to outliers and overfitting. Value of 1 means that we will consider only the closest (or nearest) neighbor to predict the class for our testing sample, and in majority of the cases it will lead to overfitting.

Press enter or click to view image in full size



Consider the case here, if we choose value of K to be 1, the given training sample will get classified as dot (Class 2), instead of being classified as a cross (Class 1). So, in such cases certain optimal value of K should be chosen to get good results. Choosing the value of K to be 1, leads to formation of complex decision boundaries and hence will lead to overfitting.

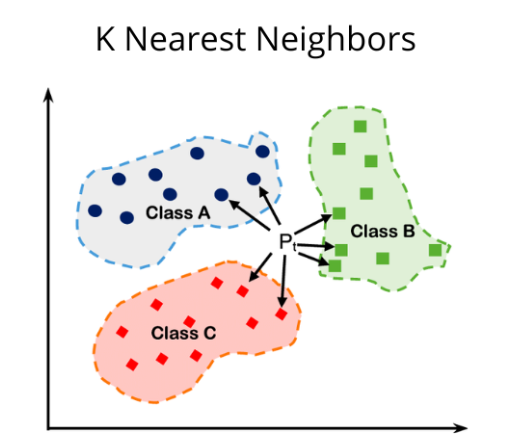
If we choose a larger K, we can get a more general idea by considering a majority vote.

Since we will be using the majority vote technique, so value of K is taken to be odd, to obtain clear result about the class of the testing data sample.

# Example :   
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn import datasets  
  
# Load dataset  
cancer = datasets.load\_breast\_cancer()  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(cancer.data,   
 cancer.target, test\_size=0.2, random\_state=0)  
  
# Initialize the KNN classifier  
clf = KNeighborsClassifier(n\_neighbors=5)  
clf.fit(X\_train, y\_train)  
  
# Evaluate the model  
print(f'Accuracy: {clf.score(X\_test, y\_test)}')

**How Does KNN Work?**

In the vast realm of machine learning algorithms, few techniques stand as versatile and intuitive as the K-nearest neighbors (KNN) algorithm. If you’re seeking a powerful tool for pattern recognition, classification, and regression tasks, KNN offers a straightforward yet effective approach that leverages the power of proximity.



So, what exactly is the K-nearest neighbors algorithm? At its core, KNN is a ***supervised machine learning algorithm*** that excels at classification and regression tasks. The KNN algorithm works based on the idea that similar things are closer to each other.

Imagine you have a dataset of different flowers, each described by their petal length and width. You also have labels indicating whether each flower is a rose or a daisy. Now, you encounter a new flower and want to determine its type. This is where KNN comes into play. Now, In this flower example, KNN looks at the flower’s nearest neighbors to decide its type. Let’s say we set K = 3, meaning we’ll consider the three closest neighbors to make our prediction. To determine the type of the new flower using KNN, here’s what we’ll do:

1. **Measure the distance:** Calculate the distance between the new flower and all the flowers in the dataset. In our case, we can use the Euclidean distance, which is like measuring the straight-line distance between two points. The distance is computed based on the petal length and width.
2. **Find the K nearest neighbors:** Identify the K flowers with the shortest distances to the new flower. These are the K nearest neighbors. For instance, if K = 3, we select the three flowers that are closest to our new flower.
3. **Majority voting:** Among the K nearest neighbors, count how many are roses and how many are daisies. Whichever type has the majority becomes our prediction for the new flower. For example, if two neighbors are roses and one is a daisy, we predict that the new flower is a rose.

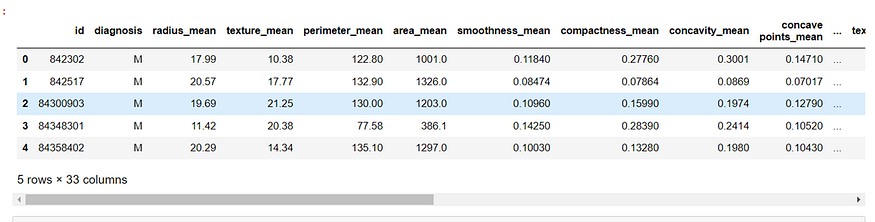
So, this is the idea behind How KNN works.

Now we will learn how to implement KNN in our code by taking [breast cancer detection dataset from Kaggle](https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data?resource=download) in our Jupyter Notebook. Let’s start:

import pandas as pd  
import numpy as np  
  
df=pd.read\_csv('data.csv')  
df.head(5)

The dataset looks like:-

Press enter or click to view image in full size



Since, in the dataset two columns are unnecessary occupy the memory, so we need to drop those column.

df.drop(columns=['id','Unnamed: 32'],inplace=True)  
df.head()  
from sklearn.model\_selection import train\_test\_split  
x\_train,x\_test,y\_train,y\_test=train\_test\_split(df.iloc[:,1:],df.iloc[:,0],test\_size=0.2,random\_state=2)

Since, in our dataset columns values are not at same scale, so we need to standardize all the columns values in order to get higher accuracy of our model. Standardization, also known as Z-score normalization, transforms the data so that it has zero mean and unit variance. This is achieved by subtracting the mean of each feature and dividing by its standard deviation. The StandardScaler class in scikit-learn performs standardization.

from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(x\_train)  
X\_test = scaler.transform(x\_test)

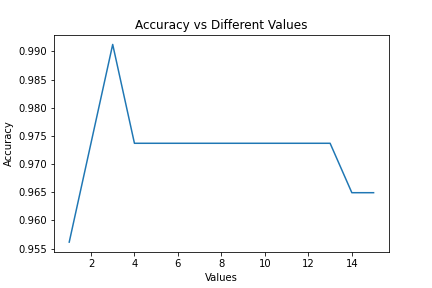
Now, we need to make object of KNeighborClassifier from scikit-learn library.

from sklearn.neighbors import KNeighborsClassifier  
knn = KNeighborsClassifier(n\_neighbors=3)  
knn.fit(X\_train,y\_train)  
from sklearn.metrics import accuracy\_score  
y\_pred = knn.predict(X\_test)  
accuracy\_score(y\_test, y\_pred)

And, I got 99.12% accuracy. Here a question arises how to select best value for **k** in order to get highest accuracy?

Answer is very simple, by performing experiment on each value. Below is the code for that:-

scores = []  
for i in range(1,16):  
 knn = KNeighborsClassifier(n\_neighbors=i)  
 knn.fit(X\_train,y\_train)  
 y\_pred = knn.predict(X\_test)  
 scores.append(accuracy\_score(y\_test, y\_pred))  
  
import matplotlib.pyplot as plt  
plt.plot(range(1,16),scores)

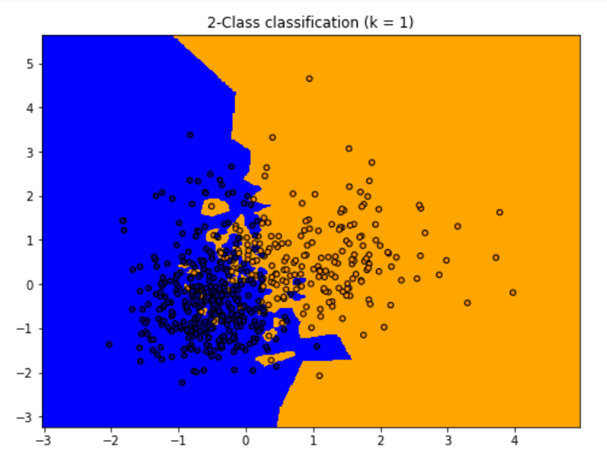


From above graph, we see that if k value is 3, then accuracy is high. so in this way we select the **k**value by experiment. [Click here for Github link](https://github.com/sachin0612/breast_cancer_detection.git).

**Overfitting and Underfitting in KNN**

1. Overfitting: Overfitting happens when a model becomes too complex, capturing noise or irrelevant patterns from the training data. In the context of KNN, overfitting can occur when we set K to a very small value, such as 1 or 2. This causes the model to become overly sensitive to local variations in the training data, leading to poor generalization to unseen data.

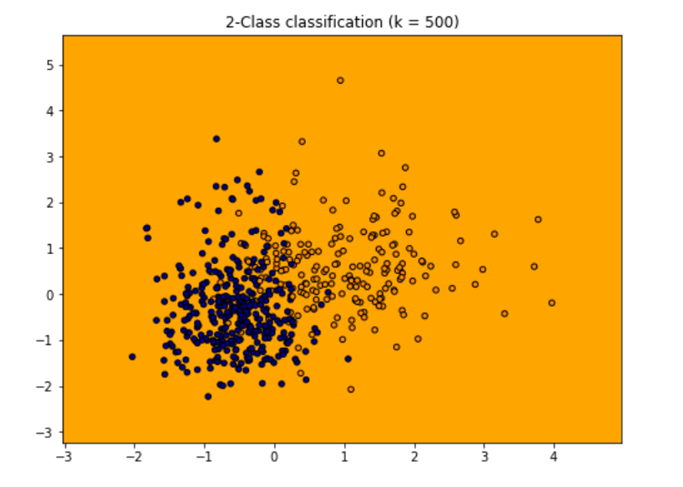
For example, consider a case where K = 1. The decision boundary between classes will follow each individual training sample closely, resulting in a model that tries to memorize the training data rather than learning the underlying patterns. This can lead to overfitting, where the model performs well on the training data but fails to generalize to new, unseen examples.



Decision boundary in case of overfitting

2. Underfitting: Underfitting occurs when a model is too simple to capture the underlying patterns in the data. In the case of KNN, underfitting can happen when we set K to a large value, such as the total number of training samples. This results in a model that is too generalized and oversimplifies the underlying patterns in the data.

For instance, if we set K to the total number of training samples, the decision boundary will be smooth and less influenced by individual samples. This might lead to underfitting, as the model may fail to capture local variations and intricate patterns present in the data.



Decision boundary in case of overfitting

**How to avoid Overfitting and Underfitting in case of KNN?**

To avoid overfitting and underfitting in KNN, it is important to choose an appropriate value for K. A small K value can lead to overfitting, while a large K value can result in underfitting. The optimal K value depends on the specific dataset and problem at hand. It is common to use techniques such as cross-validation or grid search to find the optimal K value that balances bias and variance, maximizing the model’s performance on unseen data.

**Limitations of KNN:-**

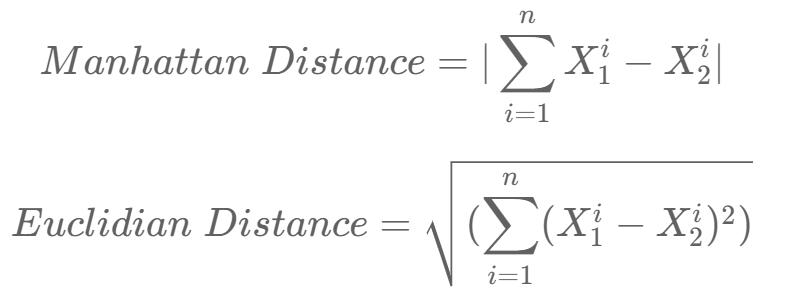
1. **Large dataset:**KNN is a lazy learning technique because in training phase KNN doing nothing, so training is fast but on time of prediction it becomes slow as large dataset comes. Let’s take an example imagine you have a dataset in which 1 lakh rows and 500 columns, in this case on prediction time your model has calculate euclidean distance from 1 lakh points in 500 dimentional dataset, due to which lots of time consume and no one likes to take much more time during prediction.
2. **Curse of Dimensionality:**The curse of dimensionality refers to the phenomenon where the feature space becomes increasingly sparse as the number of dimensions (features) grows. In high-dimensional spaces, the notion of proximity or similarity becomes less meaningful. This can negatively impact the performance of KNN, as the algorithm relies on the assumption that similar instances are close to each other. In large datasets with high-dimensional feature spaces, the accuracy and effectiveness of KNN may decrease.
3. **Irrelevant or Outlier Features:** In large datasets, it’s common to encounter features that are irrelevant or noisy, meaning they do not contribute much to the underlying patterns or have inconsistent or misleading information. KNN considers all features equally in the distance calculation, which can lead to the inclusion of irrelevant or noisy features in the decision-making process. This can result in suboptimal predictions and decreased performance.
4. **Imbalanced dataset:**In an imbalanced dataset, the majority class typically has significantly more samples than the minority class. Since KNN relies on the nearest neighbors for classification, a large number of neighbors from the majority class can overpower the neighbors from the minority class. As a result, the majority class tends to dominate the decision-making process, leading to a bias towards the majority class in the predictions.

Also, **KNN is called black box model** because of following reasons:-

1. **Lack of Explicit Rules or Parameters:** KNN does not learn explicit rules or parameters from the training data like other models such as decision trees or linear regression. Instead, it stores the training instances in memory and makes predictions based on the similarity to neighboring instances. This lack of explicit rules or parameters makes it difficult to understand the inner workings of the model.
2. **No Feature Importance or Coefficients:** KNN does not provide feature importance or coefficients that indicate the influence of each feature on the prediction. Since KNN considers all features equally in the distance calculation, it does not offer insights into which features contribute more or less to the final decision.
3. **Lack of Model Summary or Explanation:**KNN does not provide a concise summary or explanation of the model’s behavior. Unlike models such as decision trees or linear regression, which can be summarized by inspecting the learned rules or coefficients, KNN does not offer a straightforward way to summarize its behavior or reasoning.

KNN works based on **feature similarity**. To classify a new point, the algorithm calculates the “distance” between the new point and all other points in the dataset. The most common way to measure this distance is by using the **Euclidean distance and Manhattan Distance**. Once the distances are calculated, the algorithm takes a majority vote from the K-nearest points and classifies the new point accordingly.

Press enter or click to view image in full size



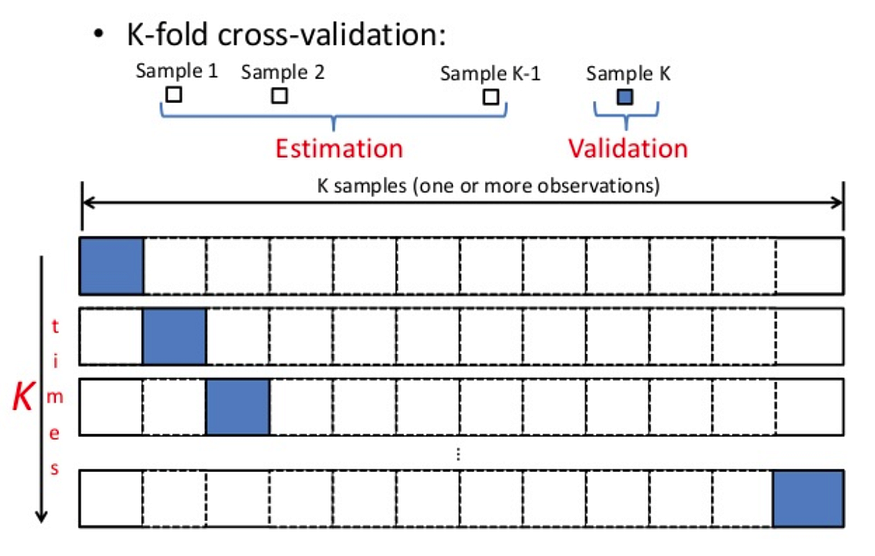
**Feature Scaling and Cross-Validation**

One important step in using KNN is **feature scaling**. If one feature is much larger in value than another, it can dominate the distance calculation. For example, if we’re using both age and income as features to classify a person’s group, the income in thousands could skew the results. To avoid this, we **normalize** the data so that each feature has equal weight.

# Example Implementation  
from sklearn.preprocessing import StandardScaler  
  
scaler = StandardScaler()  
X\_train\_scaled = scaler.fit\_transform(X\_train)  
X\_test\_scaled = scaler.transform(X\_test)

Additionally, we can use **cross-validation** to select the best value of K. This involves dividing the dataset into multiple parts, training on some, and testing on others to find the most accurate K.

Press enter or click to view image in full size



**K-Fold Cross-Validation:**

Suppose you have a dataset of 100 samples, and you choose K=5. In this case, the data is split into 5 parts, each containing 20 samples. The model is trained 5 times, each time leaving out one of the five parts as the test set. Here’s how it might look:

* **Iteration 1**: Train on folds 2, 3, 4, 5, test on fold 1.
* **Iteration 2**: Train on folds 1, 3, 4, 5, test on fold 2.
* **Iteration 3**: Train on folds 1, 2, 4, 5, test on fold 3.
* **Iteration 4**: Train on folds 1, 2, 3, 5, test on fold 4.
* **Iteration 5**: Train on folds 1, 2, 3, 4, test on fold 5.

Finally, the results of these five iterations are averaged to give an overall performance score.

from sklearn.model\_selection import KFold  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn import datasets  
from sklearn.model\_selection import cross\_val\_score  
  
# Load the breast cancer dataset  
cancer = datasets.load\_breast\_cancer()  
  
# Create KFold object with 5 folds  
kf = KFold(n\_splits=5)  
  
# Create a KNN classifier  
clf = KNeighborsClassifier(n\_neighbors=5)  
  
# Perform cross-validation and calculate accuracy  
scores = cross\_val\_score(clf, cancer.data, cancer.target, cv=kf)  
print(f"K-Fold Accuracy: {scores.mean()}")

**Stratified K-Fold:**

Suppose your dataset has 80% samples of Class 1 and 20% samples of Class 2. If you use Stratified K-Fold, each fold will have approximately 80% samples of Class 1 and 20% samples of Class 2, ensuring that the model gets a fair representation of both classes during each iteration.

Without stratification, one-fold might end up with only Class 1 samples and none from Class 2, which would not provide an accurate test of the model’s ability to generalize across both classes.

from sklearn.model\_selection import StratifiedKFold  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn import datasets  
from sklearn.model\_selection import cross\_val\_score  
  
# Load the breast cancer dataset  
cancer = datasets.load\_breast\_cancer()  
  
# Create Stratified KFold object with 5 folds  
skf = StratifiedKFold(n\_splits=5)  
  
# Create a KNN classifier  
clf = KNeighborsClassifier(n\_neighbors=5)  
  
# Perform cross-validation and calculate accuracy  
scores = cross\_val\_score(clf, cancer.data, cancer.target, cv=skf)  
print(f"Stratified K-Fold Accuracy: {scores.mean()}")

**Data Preparation:**

* Rescale Data : KNN performs better when data is on the same scale. Normalizing to [0, 1] or standardizing (for Gaussian distributions) improves accuracy.
* Handle Missing Data: Missing values can prevent distance calculations. Either remove incomplete samples or use imputation to fill missing data.
* Reduce Dimensionality : KNN works best with fewer features. In high-dimensional data, feature selection can improve performance by reducing irrelevant variables.

**✅ 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?**

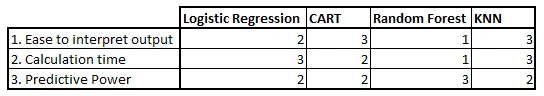
KNN Algorithm can be used for both classification and regression predictive problems. However, it is more widely used in classification problems in the industry. To evaluate any technique, we generally look at 3 important aspects:

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.

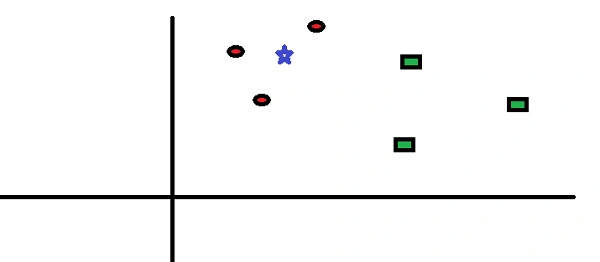
**K** parameter is critical because:

* 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.

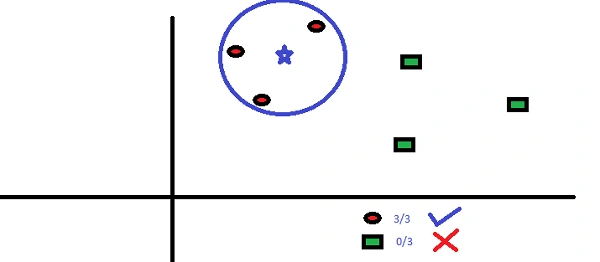
**In machine learning, noise refers to:**

* 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:

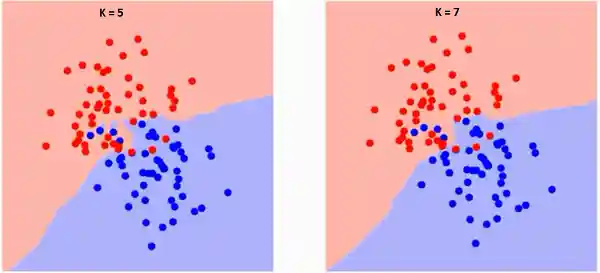


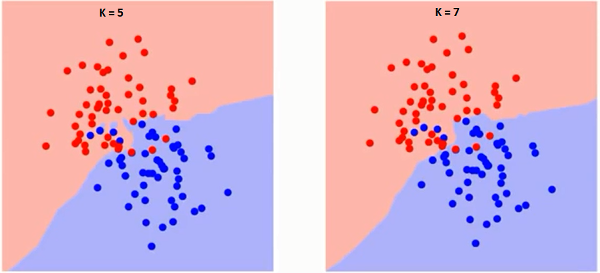
Where **N** is the number of training samples. Then try nearby values to tune.

**✅ 4. Avoid Extremes**

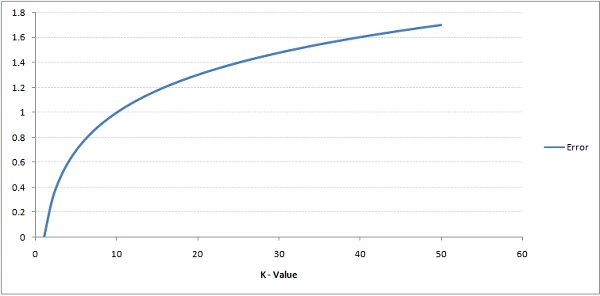
* **Too small (e.g., K=1)** → overfits, too sensitive to noise
* **Too large (e.g., K=100)** → underfits, ignores local structure

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.

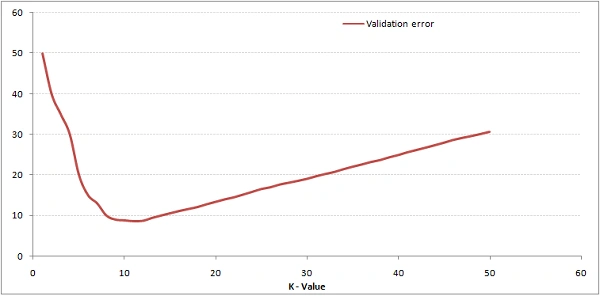




If you watch carefully, you can see that the boundary becomes smoother with increasing value of K. With K increasing to infinity it finally becomes all blue or all red depending on the total majority.  The training error rate and the validation error rate are two parameters we need to access different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

**mlextend surface plotting code –**

Great pick—mlxtend makes plotting KNN decision boundaries super easy. Below are two clean, runnable examples:

* Example A: non-linear 2-class data (make\_moons) + KNN + decision surface
* Example B: multi-class Iris (2 selected features) + KNN + decision surface

Install once if needed:

pip install mlxtend scikit-learn matplotlib

**A) KNN on moons dataset + decision boundary (2 classes)**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_moons

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score

from mlxtend.plotting import plot\_decision\_regions

# 1) Data

X, y = make\_moons(n\_samples=600, noise=0.25, random\_state=42)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=42, stratify=y

)

# 2) Model (scale -> KNN)

knn = Pipeline([

('scaler', StandardScaler()),

('knn', KNeighborsClassifier(n\_neighbors=7, weights='distance'))

])

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

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

print("Test accuracy:", accuracy\_score(y\_test, y\_pred))

# 3) Plot decision regions (mlxtend expects exactly 2 features)

plt.figure(figsize=(7, 6))

plot\_decision\_regions(X=np.vstack([X\_train, X\_test]),

y=np.hstack([y\_train, y\_test]),

clf=knn,

legend=2)

plt.title("KNN decision boundary on make\_moons (k=7, distance weights)")

plt.xlabel("Feature 1"); plt.ylabel("Feature 2")

plt.tight\_layout()

plt.show()

**B) KNN on Iris (3 classes) using two features**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score

from mlxtend.plotting import plot\_decision\_regions

# 1) Data (pick 2D for surface plotting)

iris = load\_iris()

X\_full = iris.data

y = iris.target

feature\_names = iris.feature\_names

# choose two features (e.g., petal length, petal width)

feat\_idx = [2, 3]

X = X\_full[:, feat\_idx]

chosen\_names = [feature\_names[i] for i in feat\_idx]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=0, stratify=y

)

# 2) Model

knn = Pipeline([

('scaler', StandardScaler()),

('knn', KNeighborsClassifier(n\_neighbors=9))

])

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

print("Test accuracy:", accuracy\_score(y\_test, knn.predict(X\_test)))

# 3) Plot decision regions

plt.figure(figsize=(7, 6))

plot\_decision\_regions(X=np.vstack([X\_train, X\_test]),

y=np.hstack([y\_train, y\_test]),

clf=knn,

legend=2)

plt.title("KNN decision boundary on Iris (k=9)")

plt.xlabel(chosen\_names[0]); plt.ylabel(chosen\_names[1])

plt.tight\_layout()

plt.show()

**Tips & gotchas**

* plot\_decision\_regions **requires exactly 2 features**. For high-D data, reduce to 2D first (select two informative features or apply PCA to 2 components).
* Passing a **Pipeline** is fine; mlxtend will call predict on it.
* For imbalanced data, try weights='distance' and tune n\_neighbors.
* If you really need to visualize >2 features, project with PCA:

from sklearn.decomposition import PCA

pca2 = PCA(n\_components=2)

X2 = pca2.fit\_transform(X\_high\_dim)

# then use plot\_decision\_regions with X2

Want a version that runs PCA automatically when you give any dataset, then plots the boundary on the top 2 components?

**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**.

**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) |

**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.