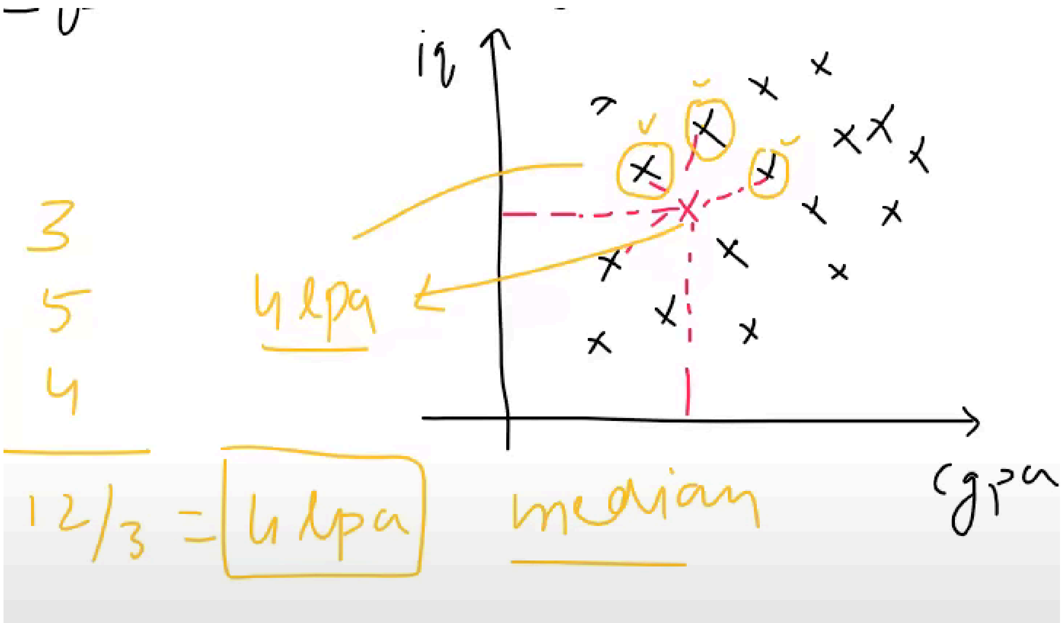
Advanced KNN

KNN Regressor

- You can apply this on regression problems.



- It finds out the nearest `k` points
- and finds the average of them

- **Type**: Supervised learning algorithm for **regression** (predicting continuous values).
- **Key Idea**: Predicts the target value for a new data point by **averaging** the target values of its `k` nearest neighbors.
- **Non-Parametric**: No assumptions about data distribution.
- **Instance-Based**: Stores the entire dataset (no explicit training phase).

- **Small k:** Low bias, high variance (sensitive to noise) → **OVERFITTING**.

- **Large k:** High bias, low variance (smoother predictions) → **UNDERFITTING**.

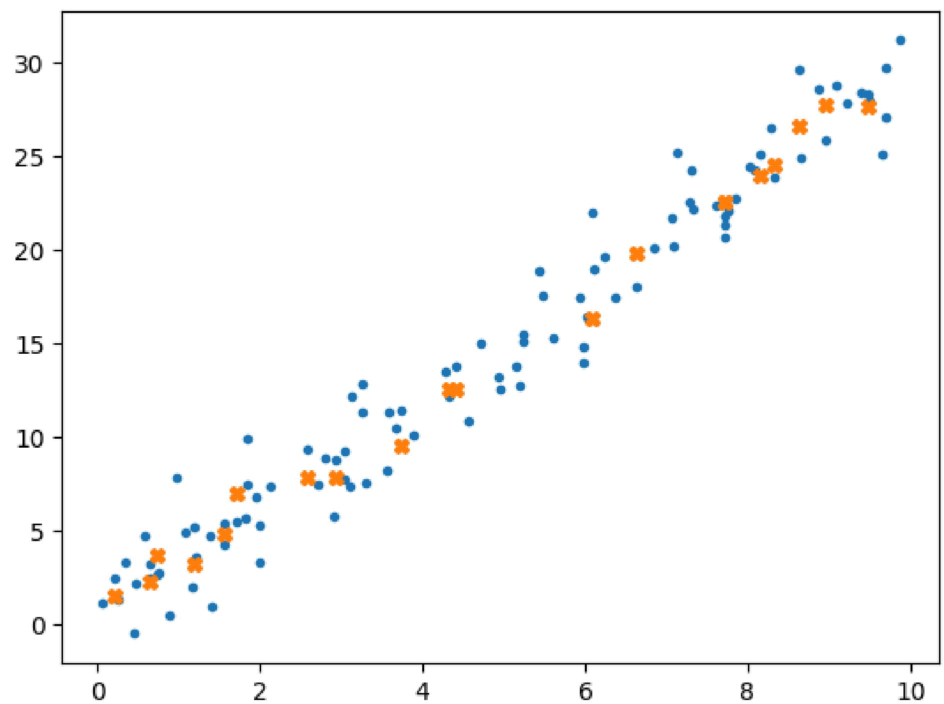
Key Parameters (scikit-learn)

Parameter	Description
`n_neighbors` (k)	Number of neighbors (default=5). Larger `k` → smoother predictions.
`weights`	`'uniform'` (all neighbors have equal weight) or `'distance'` (weight by 1/distance).
`metric`	Distance metric (`'euclidean'` , `'manhattan'` , `'minkowski'` , etc.).
`algorithm`	Algorithm for neighbor search (`'auto'` , `'ball_tree'` , `'kd_tree'` , `'brute'`).

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error

# Generate a synthetic dataset
np.random.seed(42)
X = np.random.rand(100, 1) * 10  # 100 samples, 1 feature (values between 0 a
nd 10)
y = 3 * X.squeeze() + np.random.randn(100) * 2  # Linear relation: y = 3*x + n
oise

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
e=42)

# Create a KNN Regressor with k=5 neighbors
knn_reg = KNeighborsRegressor(n_neighbors=5)
knn_reg.fit(X_train, y_train)

# Predict on the test set
y_pred = knn_reg.predict(X_test)
```

```
# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print("Test MSE:", mse)

Output:
Test MSE: 3.6046097253613296
```

```
plt.plot(X,y, '.')
plt.plot(X_test,y_pred, 'X')
```



Best Score using GridSearchCV

```
from sklearn.model_selection import GridSearchCV
param= {'n_neighbors':np.arange(1,15)}
```

```
gsv= GridSearchCV(KNeighborsRegressor(),param, cv=5,scoring='r2')
gsv.fit(X_train, y_train)
```

```
bestk = gsv.best_params_
bestk

Output:
{'n_neighbors': 9}
```

```
gsvpred=gsv.predict(X_test)
r2_score(y_test, gsvpred)

Output:
0.965868224766871
```

Hyperparameters

`n_neighbors` *: Number of neighbours*

`weights` *: {'uniform', 'distance'}*

- `'uniform'` : Uniform weights. All points in each neighborhood are weighted equally.
- `'distance'` : Closer neighbors of a query point will have a greater influence than neighbors which are further away.

Weighted KNN:

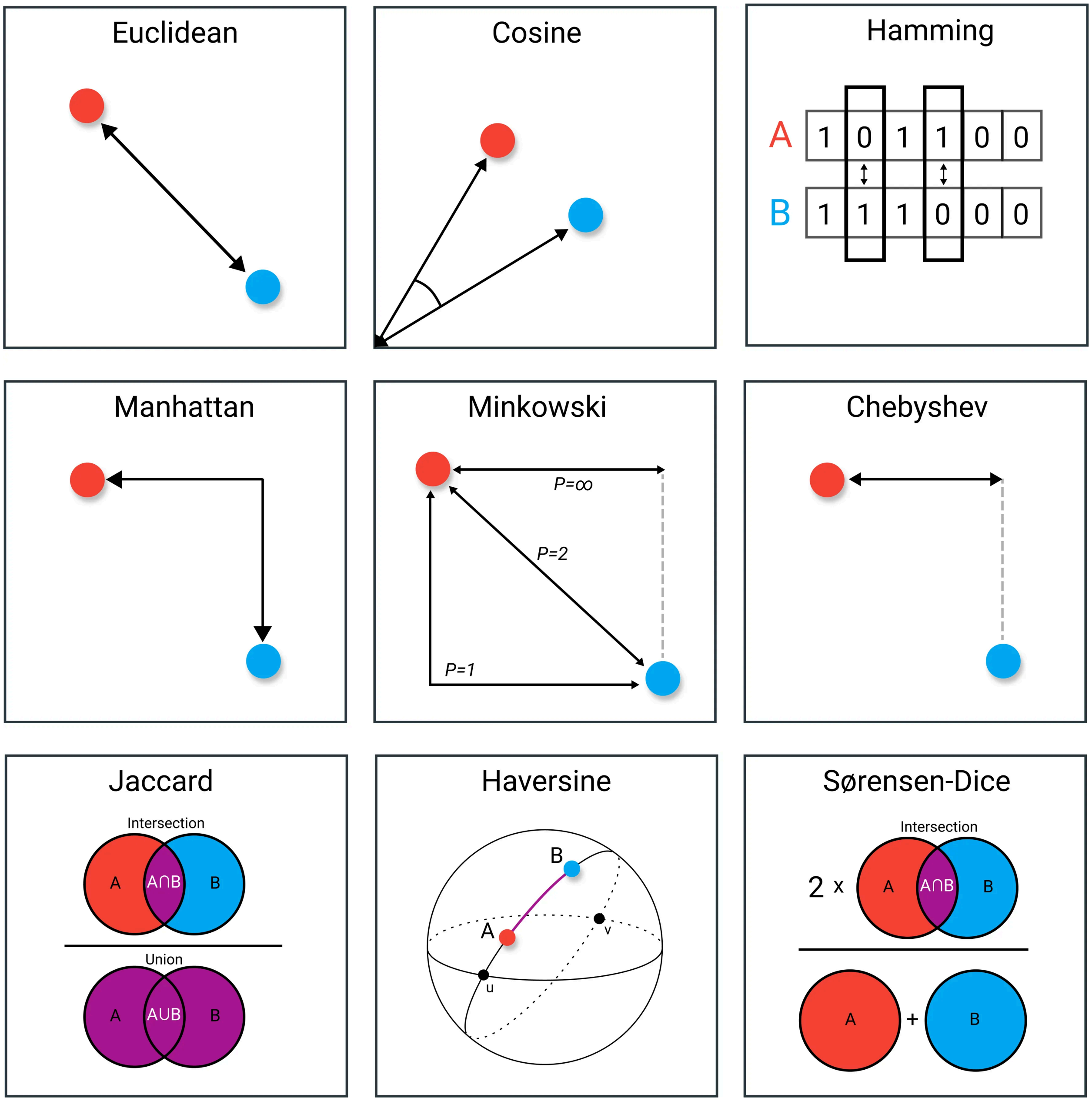
- Close neighbours=more weight

$$W = \frac{1}{distance}$$

metricstr or callable, default='minkowski'

- Metric to use for distance computation. Default is "minkowski", which results in the standard Euclidean distance when p = 2.
- When p=2 → It's Euclidian distance
- p=1 → Manhattan distance
- For higher dimension data:
  - Try p=1

Types of distances:



n_jobsint, default=None

- The number of parallel jobs to run for neighbors search. `None` means 1
- `-1` means using all processors.

algorithm{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'

Time complexity measures how the runtime of an algorithm grows as the input size increases. It's expressed using **Big O notation** (e.g., O(n), where `n` is the input size.

Time Complexity of KNN

Training Phase:

- **Time Complexity**: **O(1)**
    - KNN is a **lazy learner**, meaning it doesn't "train" a model. It simply stores the dataset.

Prediction Phase:

- **Time Complexity: O(n * d)**
    - `n`: Number of training samples.
    - `d`: Number of features.
    - For each prediction, KNN calculates distances to all training points (`O(n * d)`).
- **space complexity: O(n * d)**

Brute:

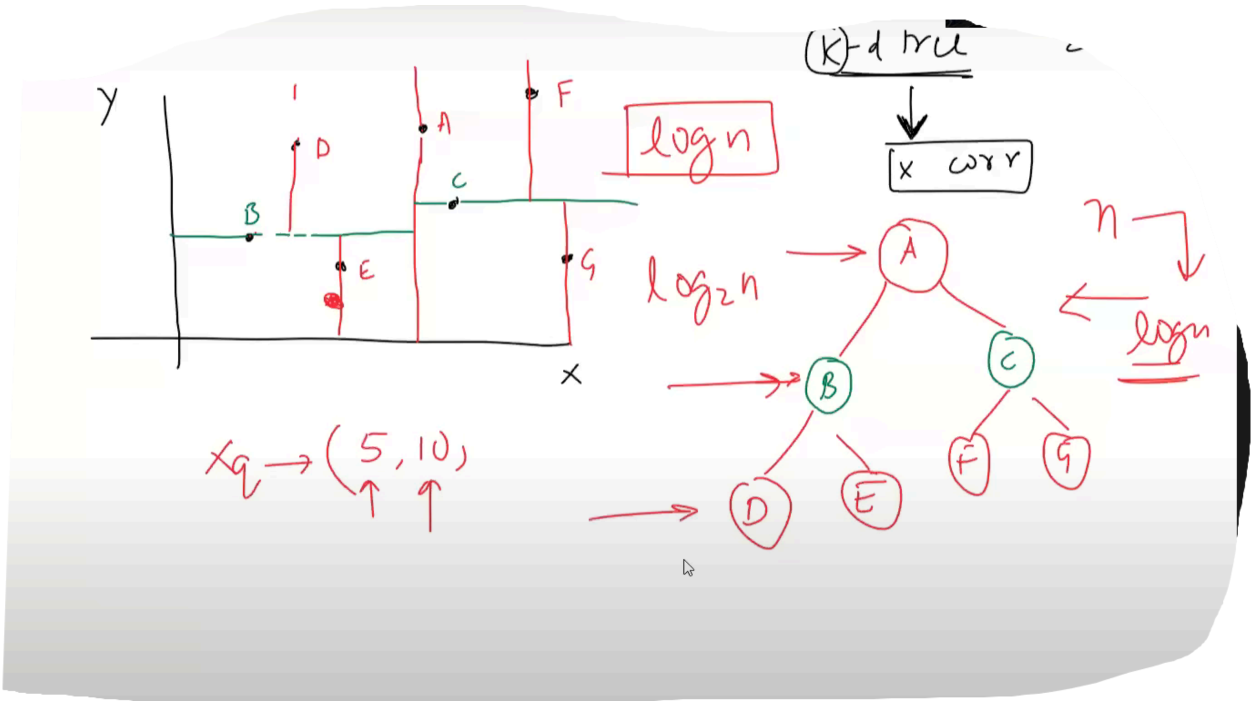
- Finding distance for each and every point
- Time consuming
- **KD-Tree** & **Ball Tree solves this problem**
- Distance stored in an array

Optimized Search:

- Using **KD-Tree** or **Ball Tree**:
    - **Time Complexity**: **O(d * log(n))**

- ◦ Reduces search time by organizing data into a tree structure.

- **KD-Tree: we form k-dimensional tree**
- Used when dimensions are 15 to 20



- It divides the points with median
- Accesses points in a specific tree

Ball Tree:

Used when dimensions are 20+

Phase	Time Complexity
Training	O(1)
Prediction	O(n * d) (brute force)
Prediction	O(d * log(n)) (KD-Tree)