**1. Euclidean Distance (Measures Straight-Line Distance)**

**Formula:**

d(A,B)=∑(Ai−Bi)2d(A, B) = \sqrt{\sum (A\_i - B\_i)^2}d(A,B)=∑(Ai​−Bi​)2​

**How it Works:**

* It calculates the straight-line (shortest) distance between two points in n-dimensional space.
* **Smaller distance = More similarity.**

**Example:**

* Comparing **two students' test scores** (as percentages):
  + **Student A:** [50, 60, 70, 80, 90]
  + **Student B:** [52, 63, 72, 78, 91]

**Application:**

* Used in **machine learning clustering (K-Means)**.
* Finding similarity between two objects in a **continuous numerical space**.

**Python Execution:**

python

CopyEdit

import numpy as np

from scipy.spatial.distance import euclidean

A = np.array([50, 60, 70, 80, 90])

B = np.array([52, 63, 72, 78, 91])

euclidean\_dist = euclidean(A, B)

print("Euclidean Distance:", euclidean\_dist)

**2. Manhattan Distance (Taxicab Distance)**

**Formula:**

d(A,B)=∑∣Ai−Bi∣d(A, B) = \sum |A\_i - B\_i|d(A,B)=∑∣Ai​−Bi​∣

**How it Works:**

* It sums up the **absolute differences** between numbers.
* Used in scenarios where **only horizontal/vertical movement** is allowed.

**Example:**

* Comparing **house prices in two cities**:
  + **City A Prices:** [200K, 220K, 250K, 270K, 300K]
  + **City B Prices:** [210K, 230K, 245K, 275K, 290K]

**Application:**

* Used in **city road network optimization**.
* Helps in **image processing for edge detection**.

**Python Execution:**

python

CopyEdit

from scipy.spatial.distance import cityblock

A = np.array([200, 220, 250, 270, 300]) # House prices in City A (in K$)

B = np.array([210, 230, 245, 275, 290]) # House prices in City B (in K$)

manhattan\_dist = cityblock(A, B)

print("Manhattan Distance:", manhattan\_dist)

**3. Minkowski Distance (Generalized Distance)**

**Formula:**

d(A,B)=(∑∣Ai−Bi∣p)1pd(A, B) = \left( \sum |A\_i - B\_i|^p \right)^{\frac{1}{p}}d(A,B)=(∑∣Ai​−Bi​∣p)p1​

**How it Works:**

* It is a **generalized version** of **Euclidean (p=2)** and **Manhattan (p=1)** distances.
* We can **tune p** to adjust sensitivity.

**Example:**

* Comparing **salary distributions** in two companies.
  + **Company A Salaries:** [50K, 60K, 70K, 80K, 100K]
  + **Company B Salaries:** [55K, 63K, 75K, 85K, 110K]
* **p=3** gives more weight to larger differences.

**Application:**

* Used in **adaptive distance metrics** in machine learning.

**Python Execution:**

python

CopyEdit

minkowski\_dist = np.sum(np.abs(A - B) \*\* 3) \*\* (1/3)

print("Minkowski Distance (p=3):", minkowski\_dist)

**4. Mahalanobis Distance (Accounts for Variance)**

**Formula:**

d(A,B)=(A−B)TS−1(A−B)d(A, B) = \sqrt{(A - B)^T S^{-1} (A - B)}d(A,B)=(A−B)TS−1(A−B)​

**How it Works:**

* Measures distance while considering **variance and correlations**.
* A small distance means numbers **belong to the same distribution**.

**Example:**

* Comparing **student marks** to a class average to find outliers.

**Application:**

* Used in **fraud detection and anomaly detection**.

**Python Execution:**

python

CopyEdit

from scipy.spatial.distance import mahalanobis

from scipy.linalg import pinv

cov\_matrix = np.cov(np.vstack([A, B]), rowvar=False)

pseudo\_inv\_cov\_matrix = pinv(cov\_matrix) # Pseudo-inverse for singular matrices

mahalanobis\_dist = mahalanobis(A, B, pseudo\_inv\_cov\_matrix)

print("Mahalanobis Distance:", mahalanobis\_dist)

**5. Chebyshev Distance (Maximum Difference)**

**Formula:**

d(A,B)=max⁡(∣Ai−Bi∣)d(A, B) = \max(|A\_i - B\_i|)d(A,B)=max(∣Ai​−Bi​∣)

**How it Works:**

* Finds the **largest** difference across dimensions.
* Good for **worst-case** scenarios.

**Example:**

* Comparing **stock market fluctuations**.

**Application:**

* Used in **logistics and supply chain** for **max risk measurement**.

**Python Execution:**

python

CopyEdit

from scipy.spatial.distance import chebyshev

chebyshev\_dist = chebyshev(A, B)

print("Chebyshev Distance:", chebyshev\_dist)

**6. Cosine Similarity (Vector Matching)**

**Formula:**

cos⁡(θ)=A⋅B∣∣A∣∣∣∣B∣∣\cos(\theta) = \frac{A \cdot B}{||A|| ||B||}cos(θ)=∣∣A∣∣∣∣B∣∣A⋅B​

**How it Works:**

* Measures how similar two **numerical vectors** are.

**Example:**

* Comparing **two customers' spending patterns**.

**Application:**

* Used in **text analysis (TF-IDF, embeddings)**.

**Python Execution:**

python

CopyEdit

from sklearn.metrics.pairwise import cosine\_similarity

cosine\_sim = cosine\_similarity(A.reshape(1, -1), B.reshape(1, -1))[0][0]

print("Cosine Similarity:", cosine\_sim)

**7. Pearson Correlation Coefficient**

**Formula:**

r=∑(Xi−Xˉ)(Yi−Yˉ)∑(Xi−Xˉ)2∑(Yi−Yˉ)2r = \frac{\sum (X\_i - \bar{X}) (Y\_i - \bar{Y})}{\sqrt{\sum (X\_i - \bar{X})^2} \sqrt{\sum (Y\_i - \bar{Y})^2}}r=∑(Xi​−Xˉ)2​∑(Yi​−Yˉ)2​∑(Xi​−Xˉ)(Yi​−Yˉ)​

**How it Works:**

* Measures **linear relationships** between numbers.

**Example:**

* Comparing **advertising spend vs. sales revenue**.

**Application:**

* Used in **financial market analysis**.

**Python Execution:**

python

CopyEdit

from scipy.stats import pearsonr

pearson\_corr, \_ = pearsonr(A, B)

print("Pearson Correlation:", pearson\_corr)

**8. Spearman’s Rank Correlation**

**Formula:**

ρ=1−6∑di2n(n2−1)\rho = 1 - \frac{6 \sum d\_i^2}{n(n^2 - 1)}ρ=1−n(n2−1)6∑di2​​

**How it Works:**

* Measures **monotonic relationships** (rank similarity).

**Example:**

* Comparing **student rankings across different tests**.

**Application:**

* Used in **social science research**.

**Python Execution:**

python

CopyEdit

from scipy.stats import spearmanr

spearman\_corr, \_ = spearmanr(A, B)

print("Spearman Rank Correlation:", spearman\_corr)

**Final Thoughts**

Each algorithm is useful for **different numeric comparison scenarios**:

* **Euclidean, Manhattan, Minkowski** → Raw distance
* **Mahalanobis, Chebyshev** → Outlier detection
* **Cosine, Pearson, Spearman** → Relationship measurement

| **Algorithm** | **Best For** | **Interpretation** |
| --- | --- | --- |
| **Euclidean Distance** | Simple numeric closeness | Smaller value = more similar |
| **Manhattan Distance** | Absolute value comparison | Smaller value = more similar |
| **Minkowski Distance** | Customizable closeness | Similar to Euclidean but tunable |
| **Mahalanobis Distance** | Statistical similarity | Close to 0 = Same distribution |
| **Chebyshev Distance** | Maximum deviation | Smaller value = less deviation |
| **Cosine Similarity** | Ratio-based similarity | Closer to 1 = More similar |
| **Jaccard Similarity** | Matching numeric sets | Closer to 1 = More overlap |
| **Pearson Correlation** | Linear relationships | Closer to 1 = Strong correlation |
| **Spearman Correlation** | Ranking similarity | Closer to 1 = Similar ranks |
| **DTW** | Numeric sequence comparison | Lower = More similar sequence |

**Rolling Hash and Rabin-Karp Algorithm**

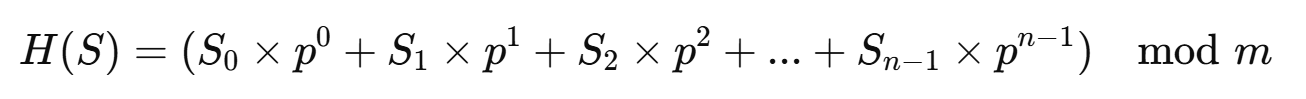
**Rolling Hash**

A **rolling hash** is a hashing technique where the hash value of a substring can be efficiently computed from the hash value of the previous substring. This allows for fast sliding window calculations in string searching algorithms.

A **common rolling hash function** is based on **polynomial hashing**, which represents a string as a polynomial and computes its hash using modular arithmetic.

**Polynomial Rolling Hash Function**

Given a string SSS of length nnn, we can compute the hash using:



* Si is the ASCII value of the character at index i.
* p is a prime number (e.g., 31 or 53) used as the base.
* m is a large prime number (modulus) to prevent overflow.
* The hash can be updated in **O(1)** time when sliding over the string.

**Efficiently Computing Rolling Hash**

**A black text on a white background

AI-generated content may be incorrect.**

This allows for efficient sliding window computations in **O(1)** time.

**Rabin-Karp Algorithm**

**Rabin-Karp** is a string-searching algorithm that uses rolling hash for pattern matching in **O(n + m)** on average.

**Algorithm Steps**

1. Compute the **hash value** of the pattern P.
2. Compute the **hash value** of the first window of text T.
3. Slide the window over the text, updating the hash using the rolling hash function.
4. If the hash matches, compare character by character to confirm the match.

**Time Complexity**

* **Best/Average Case:** O(n+m)O(n + m (hash collisions are rare).
* **Worst Case:** O(n⋅m) (all substrings have hash collisions).

def rabin\_karp(text, pattern, p=31, m=10\*\*9 + 9):

    n, m\_len = len(text), len(pattern)

    if m\_len > n:

        return []  # Pattern is longer than text, no matches possible

    p\_pow = [1] \* max(n, m\_len)

    # Precompute powers of p

    for i in range(1, len(p\_pow)):

        p\_pow[i] = (p\_pow[i-1] \* p) % m

    # Compute hash for pattern and initial window of text

    pattern\_hash = 0

    text\_hash = 0

    for i in range(m\_len):

        pattern\_hash = (pattern\_hash + (ord(pattern[i]) - ord('a') + 1) \* p\_pow[i]) % m

        text\_hash = (text\_hash + (ord(text[i]) - ord('a') + 1) \* p\_pow[i]) % m

    matches = []

    # Check first window

    if pattern\_hash == text\_hash and text[:m\_len] == pattern:

        matches.append(0)

    # Rolling hash for subsequent windows

    for i in range(1, n - m\_len + 1):

        # Remove leftmost character

        text\_hash = (text\_hash - (ord(text[i - 1]) - ord('a') + 1)) % m

        # Divide by p (equivalent to shifting window left)

        text\_hash = (text\_hash \* pow(p, -1, m)) % m

        # Add new character

        text\_hash = (text\_hash + (ord(text[i + m\_len - 1]) - ord('a') + 1) \* p\_pow[m\_len - 1]) % m

        # If hash matches, check substring

        if text\_hash == pattern\_hash and text[i:i + m\_len] == pattern:

            matches.append(i)

    return matches

# Example usage

text = "adabracadabra"

pattern = "abra"

print(rabin\_karp(text, pattern))  # Output: [7]

from collections import defaultdict

class QGramIndex:

    def \_\_init\_\_(self, q=3):

        self.q = q  # Define q-gram size

        self.index = defaultdict(set)  # Dictionary to store q-gram index

    def build\_index(self, words):

        """Builds q-gram index for a list of words."""

        for word in words:

            qgrams = self.get\_qgrams(word)

            for qgram in qgrams:

                self.index[qgram].add(word)

    def get\_qgrams(self, word):

        """Extracts q-grams from a given word."""

        padded\_word = f"${word}$"  # Padding for start & end detection

        return {padded\_word[i: i + self.q] for i in range(len(padded\_word) - self.q + 1)}

    def search(self, query):

        """Finds candidate words that share q-grams with the query."""

        qgrams = self.get\_qgrams(query)

        candidates = set()

        for qgram in qgrams:

            if qgram in self.index:

                candidates.update(self.index[qgram])

        return candidates

# Example usage

qgram\_index = QGramIndex(q=3)

words = ["banana", "bandana", "cabana", "anaconda"]

qgram\_index.build\_index(words)

query = "ban"

print(qgram\_index.search(query))  # Expected output: Words containing "ban"