UNIT – 4

RANDOMIZED AND PROBABILISTIC ALGORITHMS

In many real-world data science problems:

- Data is huge
- Exact computation is slow or impossible
- Approximate answers are acceptable

Randomized and probabilistic algorithms use randomness and probability to produce fast, memory-efficient, and scalable solutions with high accuracy.

They are widely used in:

- Big data analytics
- Recommendation systems
- Document similarity
- Web search
- Streaming data

Randomized Algorithms

A Randomized Algorithm is an algorithm that uses random numbers (random choices) during its execution to influence its behavior.
Because of randomness:

- The same input may produce different execution paths
- The algorithm's performance or output is probabilistic

Randomized algorithms are especially useful in data science and big data, where deterministic algorithms may be too slow or memory-intensive.

Why Randomized Algorithms Are Needed

Deterministic algorithms:

- Always follow the same steps
- Can perform poorly in worst-case scenarios

Randomized algorithms:

- Avoid worst-case inputs
- Are often simpler, faster, and more scalable
- Give highly accurate results with high probability

Key Characteristics

- Uses random bits or random numbers
- Output or runtime is probabilistic
- Performance measured using expected value
- Often one-pass and memory-efficient

Types of Randomized Algorithms

Randomized algorithms are mainly classified into **two types**:

- Monte Carlo Algorithms
- Las Vegas Algorithms

Monte Carlo Algorithms

Definition

- Runtime is fixed
- Output may be incorrect with a small probability
- Accuracy improves with more iterations

Characteristics

- Fast
- Approximate
- Probabilistic correctness

Key Points

- Error probability can be reduced by repeating the algorithm
- Used when approximate results are acceptable

Example

- Estimating π

- Probabilistic primality test
- Approximate counting

Advantages

- Very fast
- Easy to implement

Disadvantages

- Small chance of incorrect output

Las Vegas Algorithms

Definition

- Output is always correct
- Runtime is random

Key Points

- Uses randomness to improve average performance
- No compromise on correctness

Example

- Randomized QuickSort
- Randomized Selection Algorithm

Advantages

- Guaranteed correctness
- Efficient on average

Disadvantages

- Execution time varies

Performance Analysis of Randomized Algorithms

Expected Time Complexity

Instead of worst-case analysis, we analyze:

$$E[T(n)] = \text{Expected running time}$$

Example

Randomized QuickSort:

- Worst case: $O(n^2)$
- Expected case: O(n log n)

Advantages of Randomized Algorithms

✓ Avoid adversarial inputs
✓ Simple design
✓ Scalable to big data
✓ Efficient for streaming data

Disadvantages

✕ Output not always deterministic
✕ Slight probability of error
✕ Harder to debug

Applications in Data Science

- Sampling large datasets
- Recommendation systems
- Document similarity
- Approximate query processing
- Streaming algorithms

Real-World Example

Google Search

- Uses randomized algorithms for indexing, ranking, and similarity detection

Key Difference from Monte Carlo

Monte Carlo	**Las Vegas**

Fixed time	Random time
May be wrong	Always correct

Probabilistic Counting – Flajolet-Martin Algorithm

Problem

Count the number of **distinct elements** in a massive data stream.

Example:

- Unique users on a website
- Unique IP addresses
- Unique words in logs

Flajolet-Martin Algorithm (FM)

Introduction

In many data science applications, we need to count the number of distinct elements in a very large dataset or data stream.

Examples:

- Number of unique users visiting a website
- Number of distinct IP addresses in network logs
- Number of unique search queries

Storing all elements is memory-expensive.
The Flajolet–Martin (FM) Algorithm provides an efficient probabilistic solution.

Problem Statement

Given a data stream:

$$x_1, x_2, x_3, \dots, x_n$$

Estimate the number of distinct elements using:

- One pass
- Very small memory
- Acceptable approximation

Key Idea Behind Flajolet–Martin Algorithm

The algorithm is based on:

- Hashing
- Probability theory
- Trailing zeros in binary numbers

Core Observation

For a uniformly random hash value:

- Probability that a number ends with k trailing zeros is:

$$P = \frac{1}{2^k}$$

More distinct elements → higher chance of seeing more trailing zeros.

Algorithm Working (Step-by-Step)

Step 1: Hashing

Apply a uniform hash function to each stream element.

Example:

$$h(x) = \text{binary hash value}$$

Step 2: Count Trailing Zeros

For each hashed value:

- Count number of trailing zeros (from right)

Example:

Hash Value (Binary)	Trailing Zeros
101000	3
110100	2
100000	5

Step 3: Maintain Maximum

Keep track of:

$$R = \max(\text{trailing zeros})$$

Step 4: Estimate Distinct Count

Estimated number of distinct elements:

$$\boxed{2^R}$$

Example

Input Stream

$$\{a, b, c, d, e, f\}$$

Hashed Binary Values

Element	Hash Value	Trailing Zeros
a	101100	2
b	110000	4
c	100010	1
d	111000	3

Maximum trailing zeros:

$$R = 4$$

Estimated Distinct Count

$$2^4 = 16$$

(Actual count = 6 → approximate result)

Accuracy Improvement Techniques

Problem

Single hash function → **high variance**

Solution

Use:

- Multiple hash functions
- Average or median of estimates

Improved Estimate

$$\text{Estimated Count} = \text{Average of } 2^{R_i}$$

This significantly improves accuracy.

Advantages

✓ Very low memory usage
✓ One-pass algorithm
✓ Suitable for data streams
✓ Extremely fast

Limitations

✕ Approximate result
✕ Accuracy depends on hash quality
✕ Not exact for small datasets

Applications in Data Science

- Unique visitor counting
- Network traffic analysis
- Log file analytics
- Big data streaming platforms
- Database query optimization

Comparison with Exact Counting

Feature	**Exact Counting**	**Flajolet–Martin**
Memory	High	Very Low
Accuracy	100%	Approximate
Passes	Multiple	One

Scalability	Low	High

MinHash Algorithm

Introduction

The MinHash (Minimum Hashing) Algorithm is a probabilistic technique used to efficiently estimate the similarity between large sets, especially when the sets are too big to compare directly.

It is mainly used to approximate Jaccard Similarity in:

- Document similarity
- Plagiarism detection
- Recommendation systems
- Near-duplicate detection

Jaccard Similarity (Foundation of MinHash)

Definition

Jaccard Similarity between two sets $A$and B:

$$J(A,B) = \frac{|\ A \cap B\ |}{|\ A \cup B\ |}$$

Problem

Computing this directly is expensive for:

- Large documents
- Huge vocabularies
- Big data collections

MinHash solves this efficiently.

Basic Idea of MinHash

MinHash compresses a large set into a small signature such that:

- The probability that two sets have the same MinHash value
- Equals their Jaccard similarity

$$P(\text{MinHash}(A) = \text{MinHash}(B)) = J(A,B)$$

Working of MinHash Algorithm

Step 1: Shingling

- Break documents into k-shingles (substrings of length k)

Example (k = 3):
DATA SCIENCE → {DAT, ATA, TA , A S, SCI, CIE, IEN, ENC, NCE}

Step 2: Hash Functions

- Choose multiple independent hash functions

$$h_1, h_2, h_3, \dots, h_n$$

Step 3: Compute MinHash Values

For each set:

- Apply each hash function to all elements
- Select the minimum hash value

This forms a signature vector.

Step 4: Signature Comparison

- Compare signature vectors
- Count matching positions

Estimated similarity:

$$\text{Similarity} = \frac{\text{Number of matching MinHash values}}{\text{Total hash functions}}$$

Example

Two Documents

- Doc A = {a, b, c, d}
- Doc B = {b, c, e}

Jaccard Similarity

$$J = \frac{2}{5} = 0.4$$

MinHash Signatures

Hash Function	Doc A	Doc B
h_1	5	5
h_2	3	7
h_3	2	2

Matching = 2 out of 3

Estimated similarity:

$$\frac{2}{3} \approx 0.67$$

(Approximate result improves with more hash functions)

Why MinHash Works

- Hash functions simulate random permutations
- Minimum element acts as a representative
- Matching minimums reflect shared elements

Advantages

✓ Extremely memory efficient
✓ Fast similarity estimation
✓ Scalable for large datasets
✓ One-pass possible

Limitations

✕ Approximate results
✕ Needs many hash functions for accuracy
✕ Not suitable for small datasets

Applications

- Document similarity detection

- Web crawling and indexing
- Plagiarism checking
- Recommendation systems
- Near-duplicate webpage detection

MinHash vs Jaccard (Exact)

Feature	Exact Jaccard	MinHash
Accuracy	100%	Approximate
Speed	Slow	Fast
Memory	High	Low
Scalability	Limited	High

Locality-Sensitive Hashing (LSH)

Introduction

Locality-Sensitive Hashing (LSH) is a probabilistic technique used to efficiently find similar items in very large datasets.

The main idea is:

Similar objects are mapped (hashed) to the same bucket with high probability, while dissimilar objects are mapped to different buckets.

LSH avoids expensive pairwise comparisons, making it highly suitable for big data and data science applications.

Why LSH is Needed

For $n$objects:

- Exact similarity comparison → **$O(n^2)$** time
- Impractical for large datasets

LSH reduces this to sub-linear time.

Key Concept of LSH

An LSH family of hash functions satisfies:

- $P(h(x) = h(y))$is high if $x$and $y$are similar
- $P(h(x) = h(y))$is low if $x$and $y$are dissimilar

Relationship Between MinHash and LSH

- **MinHash** → creates compact signatures
- **LSH** → efficiently searches similar signatures

MinHash + LSH is widely used for document similarity.

Working of Locality-Sensitive Hashing

Step 1: Signature Generation

Convert each document/set into a MinHash signature.

Step 2: Banding Technique

- Divide the signature into b bands
- Each band has r rows

Total hash functions = $b \times r$

Step 3: Hashing Bands

- Each band is hashed into a bucket
- Documents in the same bucket are candidate pairs

Step 4: Similarity Check

- Only candidate pairs are compared using exact similarity

Probability Analysis

Probability that two documents with similarity $s$become a candidate pair:

$$P = 1 - (1 - s^r)^b$$

Where:

- s= Jaccard similarity
- r= rows per band
- b= number of bands

This creates an S-curve behavior.

Advantages

✓ Sub-linear search time
✓ Scalable to massive datasets
✓ Avoids full pairwise comparison
✓ Works well with high-dimensional data

Limitations

✗ Approximate method
✗ Parameter tuning required (b, r)
✗ False positives and false negatives possible

9. Applications

- Near-duplicate document detection
- Plagiarism detection
- Recommendation systems
- Image similarity
- Web search engines

Comparison: MinHash vs LSH

Feature	**MinHash**	**LSH**
Purpose	Signature creation	Similarity search
Output	Compact vectors	Candidate pairs
Speed	Fast	Very fast
Use Together	Yes	Yes

Applications: Document Similarity & Recommendation Engines

1. Document Similarity

What is Document Similarity?

Document similarity measures how much two documents are alike based on their content.

It is widely used to:

- Detect plagiarism
- Remove duplicate web pages
- Improve search engine ranking
- Group related documents

Challenges

- Documents are very large
- Vocabulary size is huge
- Exact comparison is computationally expensive

Solution: Probabilistic Algorithms

Algorithms Used

1. Jaccard Similarity
2. MinHash
3. Locality-Sensitive Hashing (LSH)

Working Process

Step 1: Shingling

- Break document into small chunks (k-shingles)

Example:

DATA SCIENCE → {DAT, ATA, TA , A S, SCI, CIE, IEN, ENC, NCE}

Step 2: MinHash Signature

- Apply multiple hash functions
- Store minimum hash values
- Convert large document → small signature

Step 3: LSH Bucketing

- Divide signatures into bands
- Hash each band
- Similar documents fall into the same bucket

Step 4: Similarity Detection

- Compare only candidate pairs
- Compute approximate Jaccard similarity

Advantages

✓ Very fast
✓ Memory efficient
✓ Scalable to millions of documents

Real-World Examples

- **Google Search** – near-duplicate page detection
- **Turnitin** – plagiarism detection
- **News aggregators** – grouping similar news articles

2. Recommendation Engines

What is a Recommendation Engine?

A recommendation engine suggests relevant items to users based on:

- User preferences
- Past behavior
- Similar users or items

Types of Recommendations

1. Content-Based Filtering
2. Collaborative Filtering

Role of Probabilistic Algorithms

Large datasets make exact matching impossible.

Randomized algorithms provide:

- Fast similarity computation
- Efficient user-item matching

Algorithms Used

- MinHash
- LSH
- Randomized Sampling
- Approximate Nearest Neighbors

How It Works (Example)

Step 1: Represent Users/Items

- Convert users or products into sets
- Example: Movies watched, products bought

Step 2: MinHash Signatures

- Create compact representation

Step 3: LSH Bucketing

- Similar users/items fall into same bucket

Step 4: Recommendation

- Recommend items liked by similar users

Advantages

✓ Real-time recommendations
✓ Scales to millions of users
✓ Low memory usage

Real-World Examples

- **Netflix** – movie recommendations
- **Amazon** – product suggestions
- **Spotify** – music recommendations
- **YouTube** – video suggestions