ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

Progress II presentation

Subject: Mining Massive Datasets

Instructor: Msc **Nguyễn Thành An**

Group’s member

Student ID	Full name	Email	Tasks	Complete percentage
520H0536	Lê Quốc Huy	520H0536@student.tdtu.edu.vn	Task 1, slide	100
522H0120	Nguyễn Đình Việt Hoàng	522H0120@student.tdtu.edu.vn	Slide	100
520H0523	Tăng Đại	520H0523@student.tdtu.edu.vn	Task 1, slide	100
521H0072	Nguyễn Thiên Huy	521H0072@student.tdtu.edu.vn	Task 2	100
521H0503	Trương Huỳnh Đăng Khoa	521H0503@student.tdtu.edu.vn	Task 2	100

Table of contents

Motivating problem

Consider the following problem:

We have several billion documents, and we want to identify near-duplicate or similar documents.

How can we solve this efficiently?

If we do naïve approach, we would go through each pair-wise document in the collection □ but that is too slow

We need an efficient way to cluster similar documents

□ Locality sensitive hashing

Motivating problem

High-Level Approach:

(1) Shingling:

- Capture consecutive characters/ word of k length
- Permits reordering of words in a document

(2) MinHash:

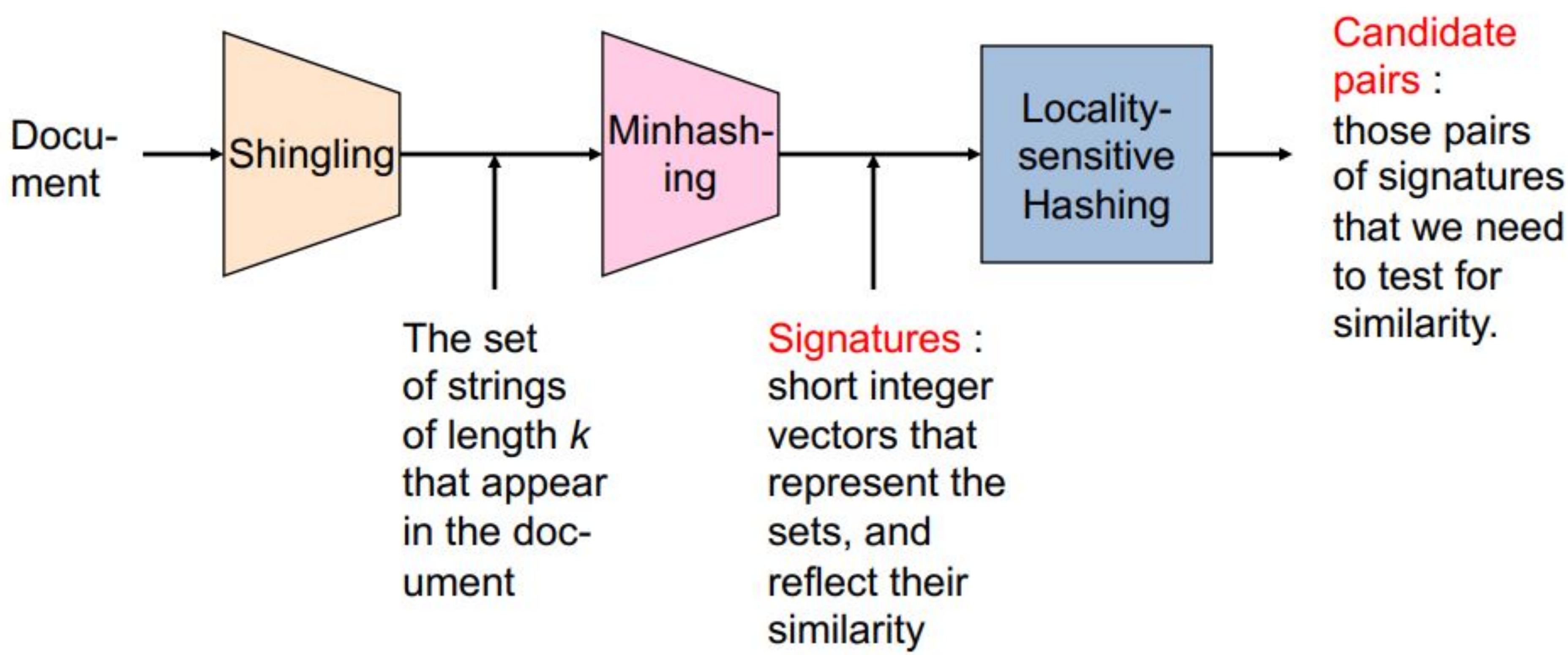
- Generate signature for shingles in a document efficiently

(3) Locality Sensitive Hash:

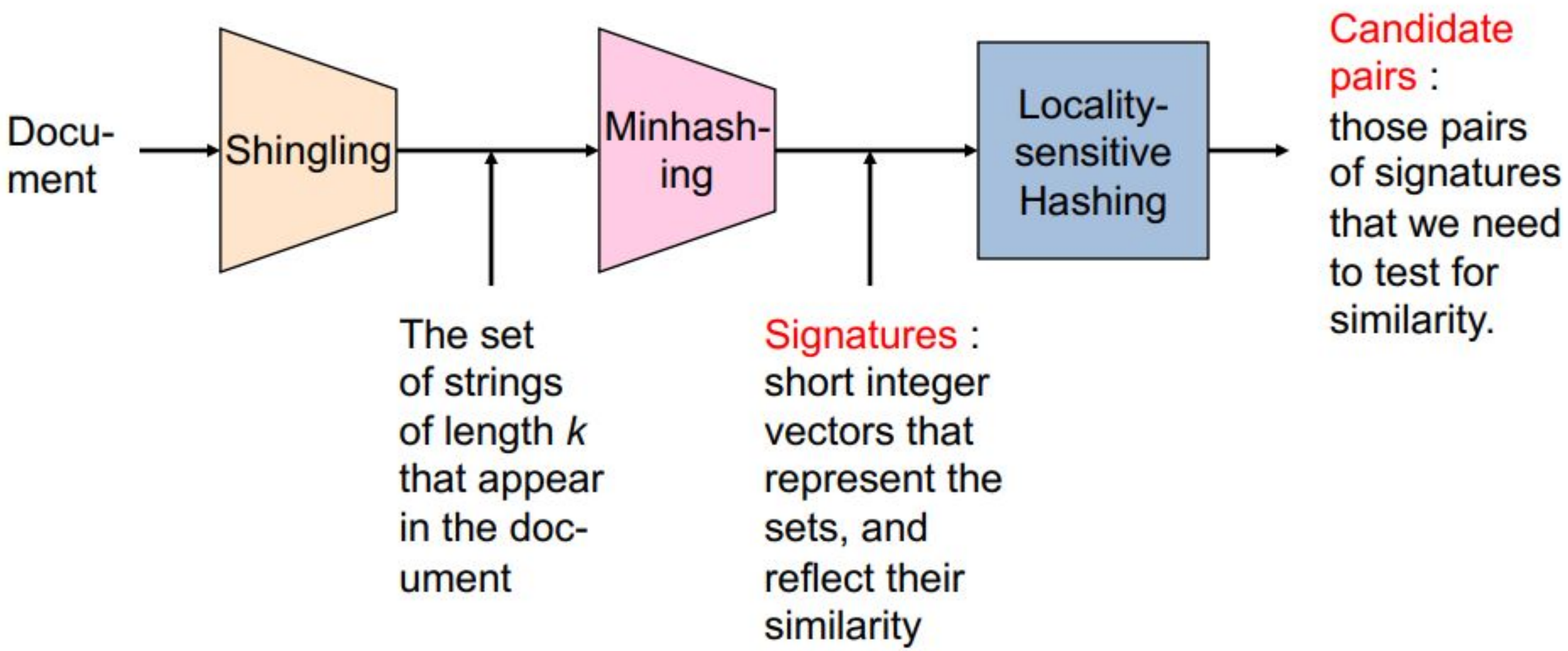
- Efficiently determine pairs of signatures that appear similar

Motivating problem

The Big Picture



Introduction to MinHashLSH algorithm



Introduction to MinHashLSH algorithm

Shingling

- k-Shingling, or simply shingling — is the process of converting a string of text into a set of 'shingles'.
- Imaging moving a window of length k down our string of text and taking a picture at each step.
- Then take collate all of those pictures to create our *set* of shingles.

Introduction to MinHashLSH algorithm

Example of Shingling with k = 2

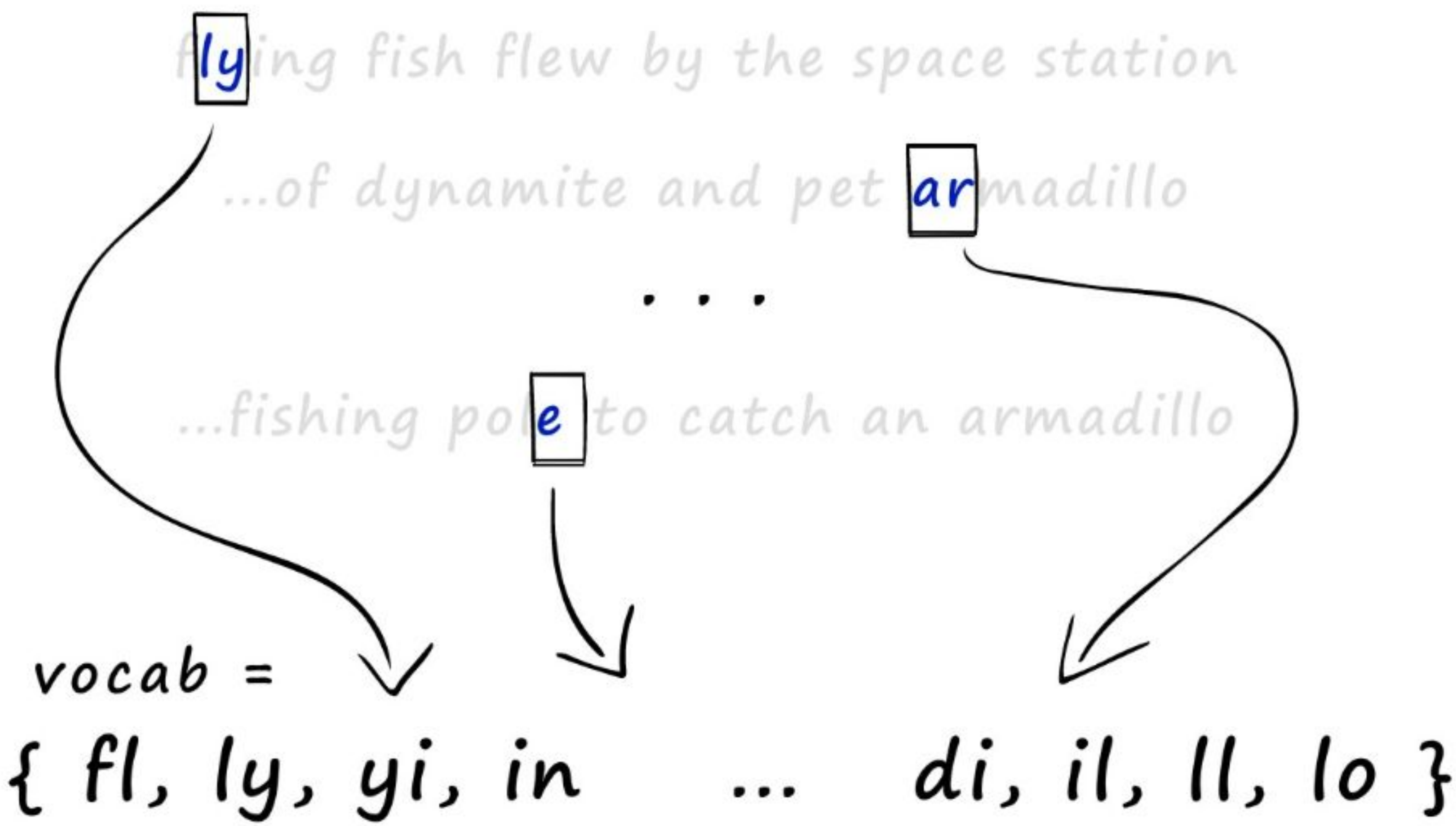
flying fish flew by the space stati**on**

{fl, ly, yi, in, ng, g_, _f, fi, is, sh
h_ ... st, ta, at, ti, io, on}

Introduction to MinHashLSH algorithm

And with this, we have our shingles. Next, we create our sparse vectors:

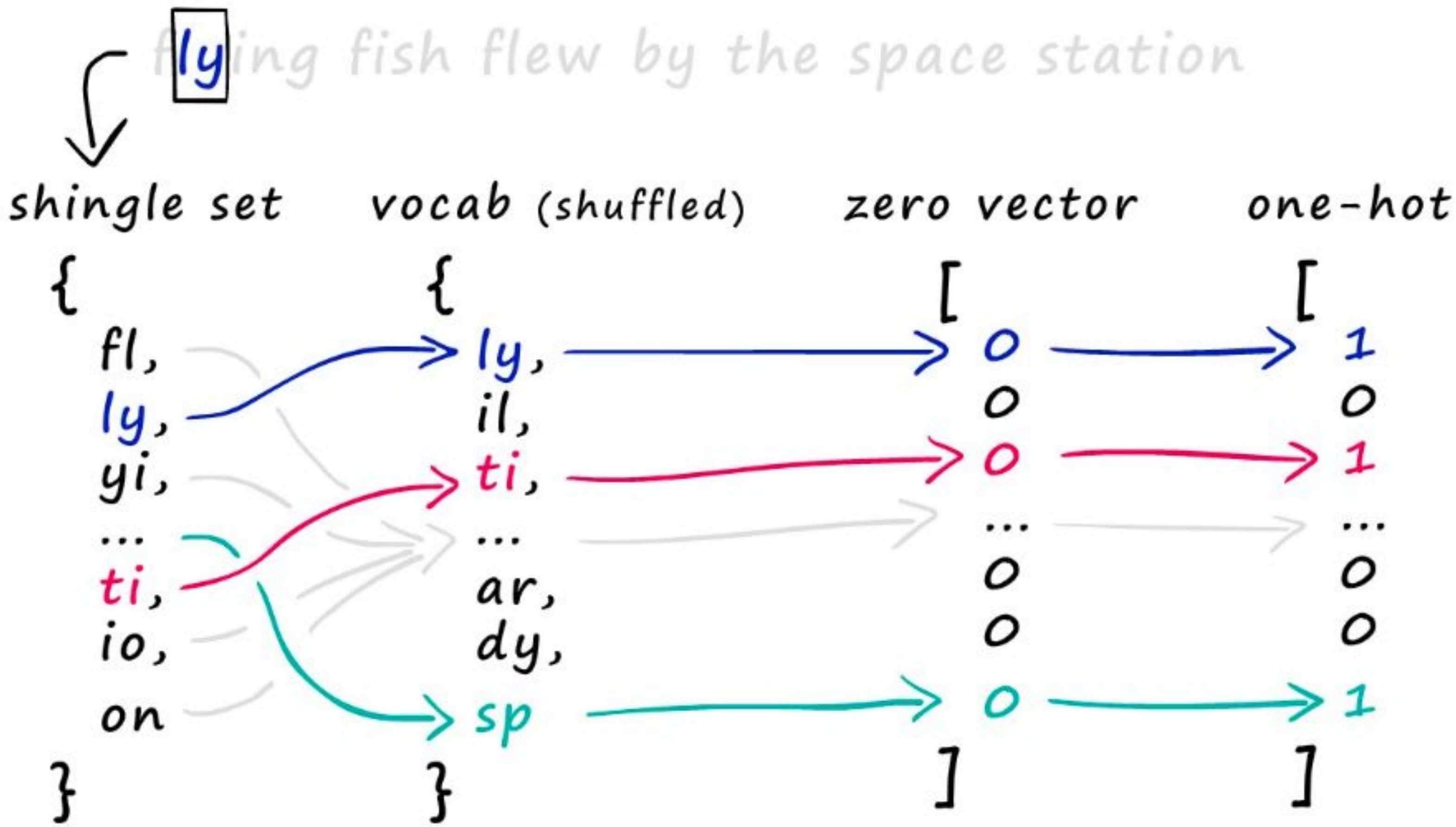
- First need to union all of our sets to create one big set containing *all* of the shingles across all of our sets — we call this the vocabulary (or vocab).



Introduction to MinHashLSH algorithm

And with this, we have our shingles. Next, we create our sparse vectors:

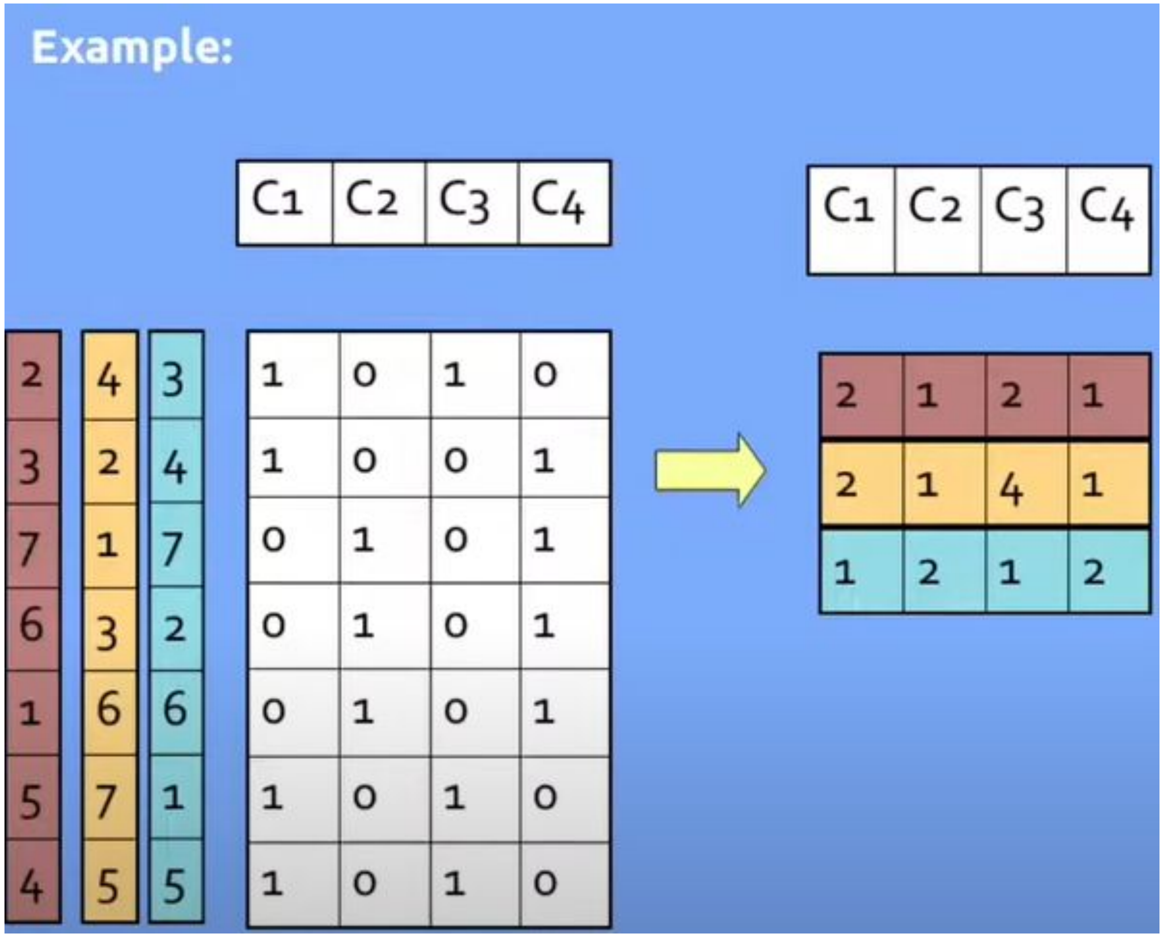
- After that use this vocab to create our sparse vector representations of each set. All we do is create an empty vector full of zeros and the same length as our vocab — then, we look at which shingles appear in our set.



Introduction to MinHashLSH algorithm

MinHashing

- Complex technique that leverages permutations of a binary string, and then identify the first value in the string that is “1”
- By using multiple permutations, a sequence can be generated acting as a “signature” for different features.



At the end of this, we produce our minhash signature — or dense vector.

Introduction to MinHashLSH algorithm

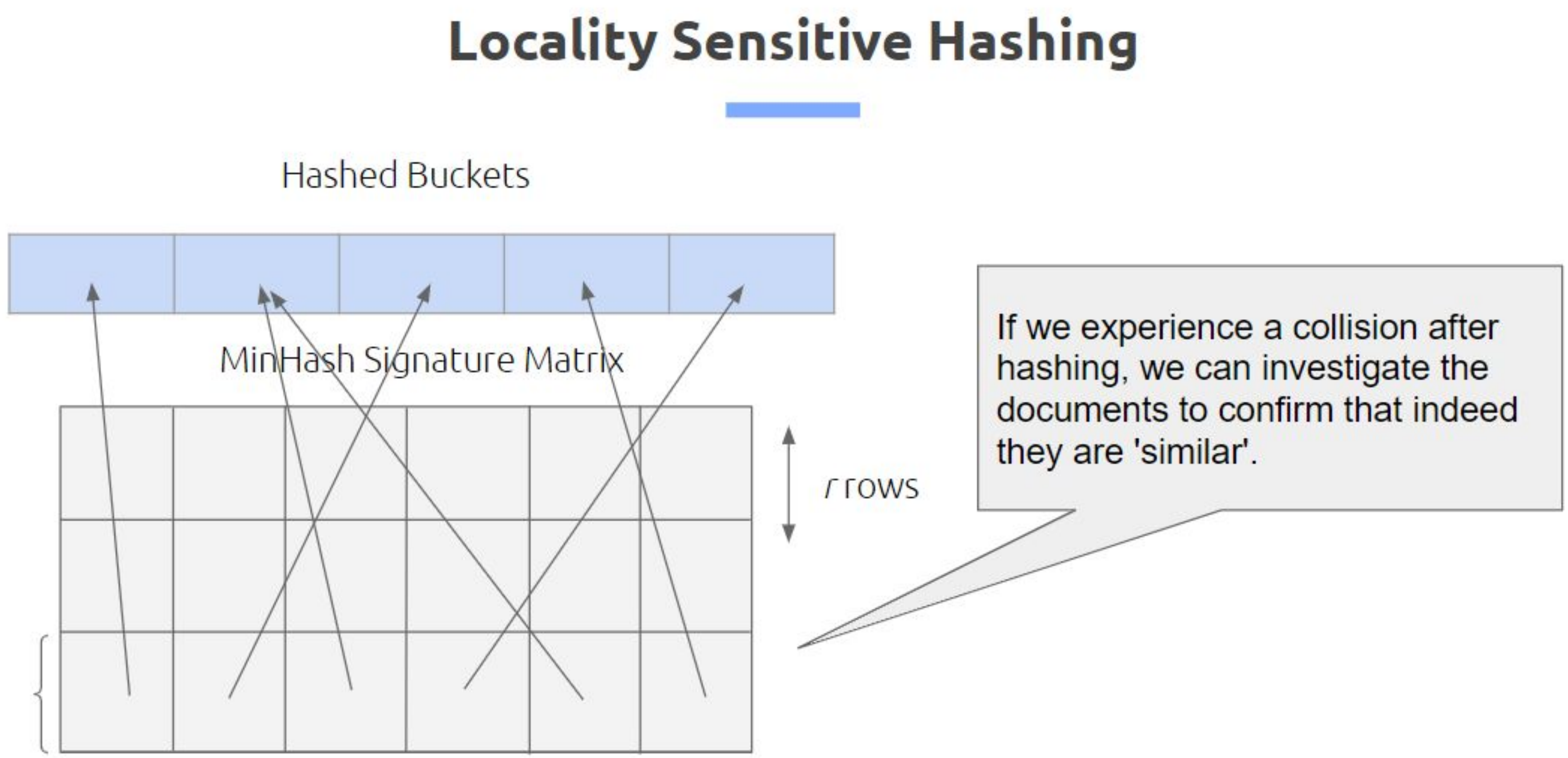
How can we find out two documents are similar?

→ We use Jaccard similarity to calculate the similarity between their signature vectors

Introduction to MinHashLSH algorithm

Locality Sensitive Hashing

- We now obtained a signature matrix, across the different permutations for each document. The goal is to identify across the matrix similar columns.
- The trick is to break down the matrix into bands, and collapse portions of each band, hash it, and then identify others that fall under the same bucket.



Introduction to MinHashLSH algorithm

Advantages of MinHashLSH algorithm

- Scalability: MinHash LSH is highly scalable and efficient for large datasets. It can handle millions or even billions of items efficiently.
- Space Efficiency: MinHash LSH reduces the dimensionality of the data, which leads to significant savings in storage space and computational resources.
- Versatility: It can be applied to various types of data, including text, images, and other high-dimensional data, making it a versatile solution for similarity search problems.
- Speed: MinHash LSH can perform similarity searches in sub-linear time, which means that the search time does not increase linearly with the size of the dataset.

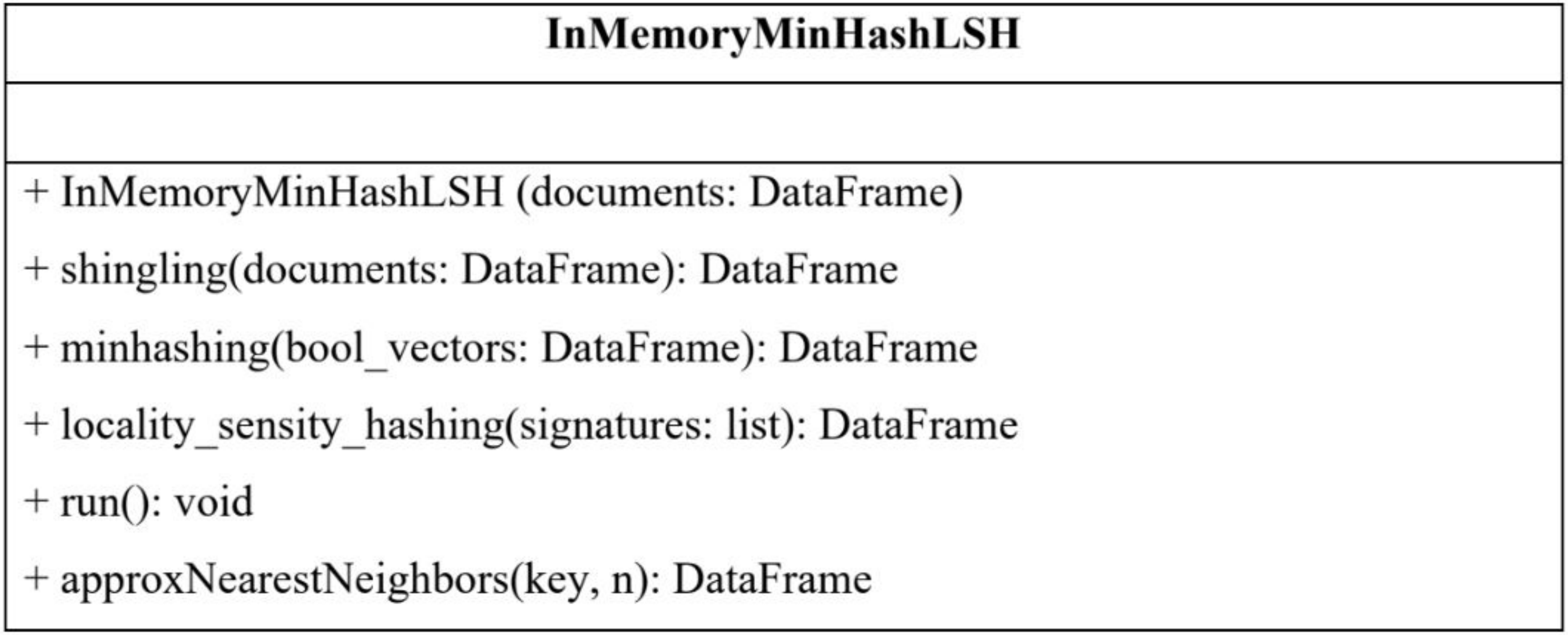
Introduction to MinHashLSH algorithm

Disadvantages of MinHashLSH algorithm

- Approximate Results: While MinHash LSH is efficient, it sacrifices accuracy for speed and scalability. The results provided by the algorithm are approximate and may not always be perfectly accurate.
- Sensitivity to Parameters: The performance of MinHash LSH can be sensitive to the choice of parameters, such as the number of hash functions and the number of hash tables. Tuning these parameters for optimal performance can be challenging.
- False Positives: Due to the nature of locality-sensitive hashing, there is a possibility of returning false positive results, where items that are not truly similar are considered similar by the algorithm.

Task 1

We use pure in-memory processing operation and encapsulated it into corresponding classes:



Pseudocode for Task 1

```
Class InMemoryMinHashLSH:
    Function shingling(self):
        shingle_size = 5
        self.documents['preprocessed_document'] = apply preprocess_document to
    each document in self.documents['document']

        Function generate_shingles(document):
            Initialize empty set shingles
            words = split document by whitespace
            For each index i from 0 to length of words - shingle_size:
                shingle = join words[i:i + shingle_size] with whitespace
                Add shingle to shingles
            Return shingles as list

        self.documents['shingles'] = apply generate_shingles to each preprocessed
    document
```

Pseudocode for Task 1

```
Function minhashing(self):
    num_perm = 128

    Function minhash_vector(shingles):
        Initialize signature as list of num_perm elements, each set to positive infinity
        For each shingle in shingles:
            For each index j from 0 to num_perm:
                hash_value = hash(shingle) XOR hash(j)
                signature[j] = min(signature[j], hash_value)
        Return signature

    self.documents['signature'] = apply minhash_vector to each shingle in
self.documents['shingles']
```

Pseudocode for Task 1

```
Function locality_sensitive_hashing(self):
    signatures = convert self.documents['signature'] to list
    num_buckets = 1000
    Initialize empty dictionary buckets

    For each signature in signatures:
        bucket_id = hash(tuple(signature)) mod num_buckets
        If bucket_id not in buckets:
            Create empty list at buckets[bucket_id]
        Append signature to buckets[bucket_id]

    self.hashed_buckets = list of tuples (bucket_id, bucket_signatures) for each
bucket_id and bucket_signatures in buckets
```

Pseudocode for Task 1

```
Function run(self):
    Call shingling method
    Call minhashing method
    Call locality_sensitive_hashing method

Function approxNearestNeighbors(self, query_document, n):
    query_shingles = set of words obtained by splitting query_document by whitespace and
removing leading and trailing white spaces
    Initialize empty list nearest_neighbors

    For each bucket_id, bucket_signatures in self.hashed_buckets:
        For each signature in bucket_signatures:
            similarity = calculate jaccard similarity between set(signature) and query_shingles
            Append (signature, similarity) to nearest_neighbors

    Sort nearest_neighbors in descending order based on similarity
    Return first n elements of nearest_neighbors

Function jaccard_similarity(self, set1, set2):
    intersection = count elements in set1 that are also in set2
    union = count unique elements in both sets
    Return intersection divided by union
```

Result of Task 1

```
query_document = "Background: Leukotoxin (Ltx) expressed by Aggregatibacter actinomyce
```

```
Top 10 most similar documents:
1. Document 342: Similarity = 0.5981308411214953
2. Document 341: Similarity = 0.15231788079470202
3. Document 5735: Similarity = 0.14393939393939395
4. Document 5043: Similarity = 0.14285714285714285
5. Document 753: Similarity = 0.1417910447761194
6. Document 5746: Similarity = 0.1417910447761194
7. Document 1324: Similarity = 0.14166666666666666
8. Document 6317: Similarity = 0.14166666666666666
9. Document 969: Similarity = 0.13846153846153847
10. Document 5962: Similarity = 0.13846153846153847
```

Task 2

We re-implement the requirements from task 1 using PySpark

+ shingling(documents: DataFrame): DataFrame
+ minhashing(bool_vectors: DataFrame): DataFrame
+ locality_sensity_hashing(signatures: list): DataFrame
+ run(): void
+ approxNearestNeighbors(key, n): DataFrame

Pseudocode for Task 2

```
function LargeScaleMinHashLSH(spark)
        self.spark ← spark
        self.documents ← null
        self.shingles ← null
        self.signatures ← null
        self.hash_buckets ← null

        function shingling(documents)
            self.shingles ← documents.select("doc_id", "text")
                .rdd.map(lambda x: (x[0], x[1].split()))
                .toDF(["doc_id", "shingles"])
```

Pseudocode for Task 2

```
function minhashing(documents, num_hash_functions=100)
    exploded_shingles ← self.shingles.select("doc_id", explode("shingles").alias("shingle"))
    hash_values ← exploded_shingles.select("shingle")
        .distinct().rdd.map(lambda x: (x[0], [hash(x[0]) % num_hash_functions for _ in range(num_hash_functions)]))
        .toDF(["shingle", "hash_values"])

    self.signatures ← exploded_shingles.join(hash_values, exploded_shingles.shingle == hash_values.shingle)
        .groupBy("doc_id").agg(collect_list("hash_values").alias("hash_values"))

function locality_sensity_hashing(documents, num_hash_buckets=10)
    self.hash_buckets ← self.signatures.rdd.flatMap(lambda x: [(tuple(h), x[0]) for h in x[1]])
        .map(lambda x: ((hash(x[0]), x[1]), x[0]))
        .groupByKey().map(lambda x: (x[0][0] % num_hash_buckets, [x[0][1]]))
        .reduceByKey(lambda x, y: x + y).collect()
function run()
    shingling(self.documents)
    minhashing(self.documents)
    locality_sensity_hashing(self.documents)
```

Pseudocode for Task 2

```
function minhashing(documents, num_hash_functions=100)
    exploded_shingles ← self.shingles.select("doc_id", explode("shingles").alias("shingle"))
    hash_values ← exploded_shingles.select("shingle")
        .distinct().rdd.map(lambda x: (x[0], [hash(x[0]) % num_hash_functions for _ in range(num_hash_functions)]))
        .toDF(["shingle", "hash_values"])

    self.signatures ← exploded_shingles.join(hash_values, exploded_shingles.shingle == hash_values.shingle)
        .groupBy("doc_id").agg(collect_list("hash_values").alias("hash_values"))

function locality_sensity_hashing(documents, num_hash_buckets=10)
    self.hash_buckets ← self.signatures.rdd.flatMap(lambda x: [(tuple(h), x[0]) for h in x[1]])
        .map(lambda x: ((hash(x[0]), x[1]), x[0]))
        .groupByKey().map(lambda x: (x[0][0] % num_hash_buckets, [x[0][1]]))
        .reduceByKey(lambda x, y: x + y).collect()
function run()
    shingling(self.documents)
    minhashing(self.documents)
    locality_sensity_hashing(self.documents)
```

Pseudocode for Task 2

```
function approxNearestNeighbors(documents, query_document, n)
    query_hash ← hash(query_document)
    for bucket_id, documents in self.hash_buckets do
        if hash(query_hash) % len(self.hash_buckets) == bucket_id then
            return documents[:n]

    return []

function jaccard_similarity(set1, set2)
    intersection ← length(set1.intersection(set2))
    union ← length(set1.union(set2))

    return intersection / union

  return self
```

Pseudocode for Task 2

```
query_document ← " "

n ← 10
results ← lsh.approxNearestNeighbors(file_content_df, query_document, n)
print("Approximate nearest neighbors:", results)
query_shingles ← set(query_document.split())

for result_doc_id in results do
    result_shingle_set ← set(file_content_df.filter(file_content_df['doc_id'] ==
result_doc_id).collect()[0]['text'])
    jaccard_similarity ← lsh.jaccard_similarity(result_shingle_set, query_shingles)
    print(jaccard_similarity)
```

Result of Task 2

```
Approximate nearest neighbors: [1453, 1485, 3407, 3873, 4681, 5306, 26, 63, 63, 100]
```

```
0.009009009009009009
0.009615384615384616
0.008849557522123894
0.01680672268907563
0.017241379310344827
0.015267175572519083
0.009433962264150943
0.018867924528301886
0.018867924528301886
0.008695652173913044
```

References

[1]“MIN-HASHING AND LOCALITY SENSITIVE HASHING Thanks to: Rajaraman and Ullman, ‘Mining Massive Datasets’ Evimaria Terzi, slides for Data Mining Course.” Accessed: Apr. 28, 2024. [Online]. Available: https://www.cs.bu.edu/~gkollios/cs660f19/Slides/minhashLSH.pdf

[2]“Locality Sensitive Hashing (LSH): The Illustrated Guide | Pinecone,” www.pinecone.io. https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/

[3]“Learn in 5 Minutes: Finding Nearest Neighbor using MinHash,” www.youtube.com. https://www.youtube.com/watch?v=GRHsg0d5X8Y (accessed Apr. 28, 2024).

[4]“Learn in 5 Minutes: Locality Sensitive Hashing (MinHash, SimHash, and more!),” www.youtube.com. https://www.youtube.com/watch?v=R-iFka68ZwM (accessed Apr. 28, 2024).

ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

Subject: Mining Massive Datasets

Thanks for your listening