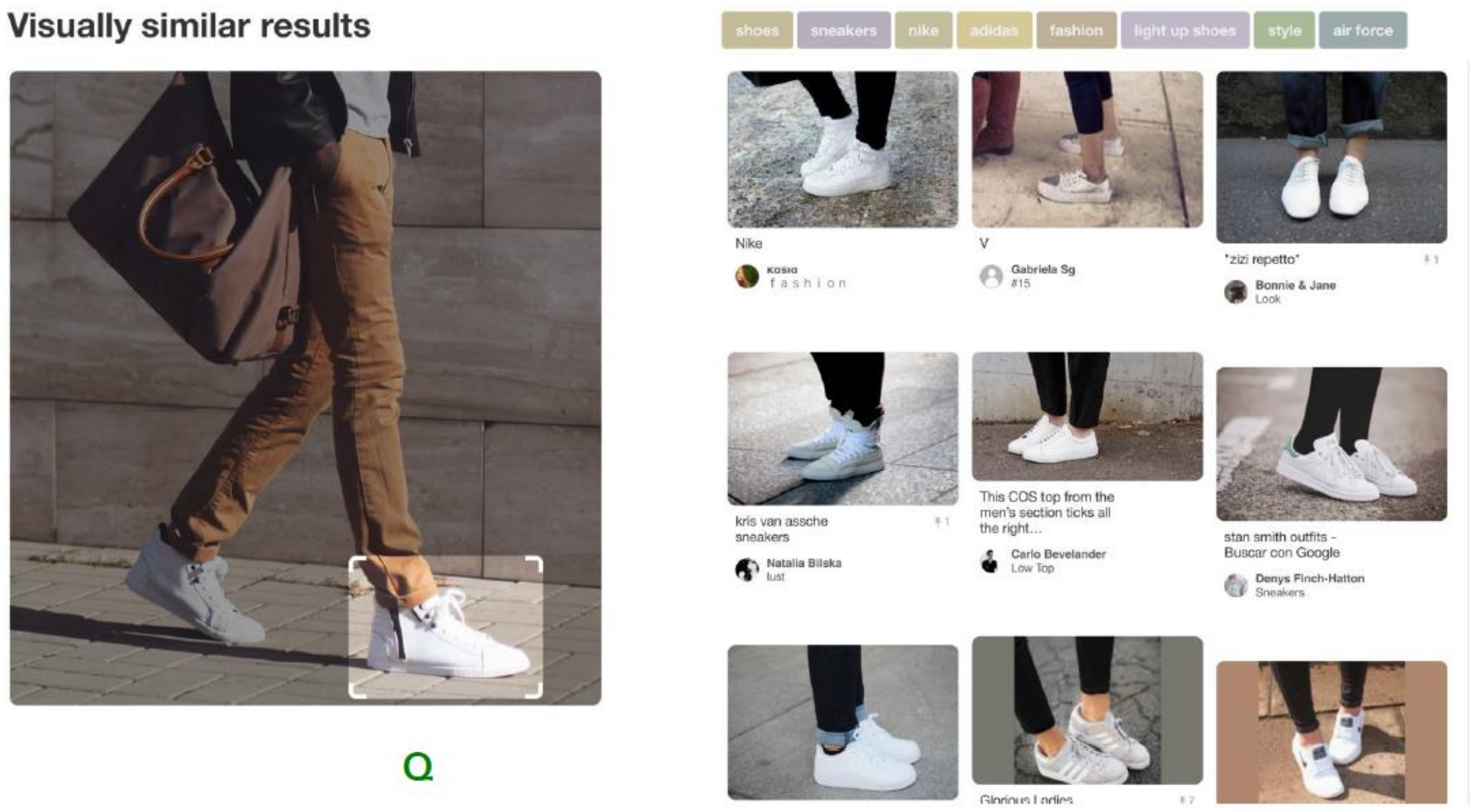
Similarity Search - LSH

Lin Guosheng
School of Computer Science and Engineering
Nanyang Technological University

Outline

- Nearest Neighbour Search (NNS) Problems
- Local Sensitive Hashing (LSH) – Random Projection
- Product Quantization (PQ)
- Inverted File Index **(non-examinable!!)**

Similarity Search Applications



https://web.stanford.edu/class/cs246/

How does it work?

Query image

Search similar images
from the database

$\xrightarrow{\hspace{6em}}$

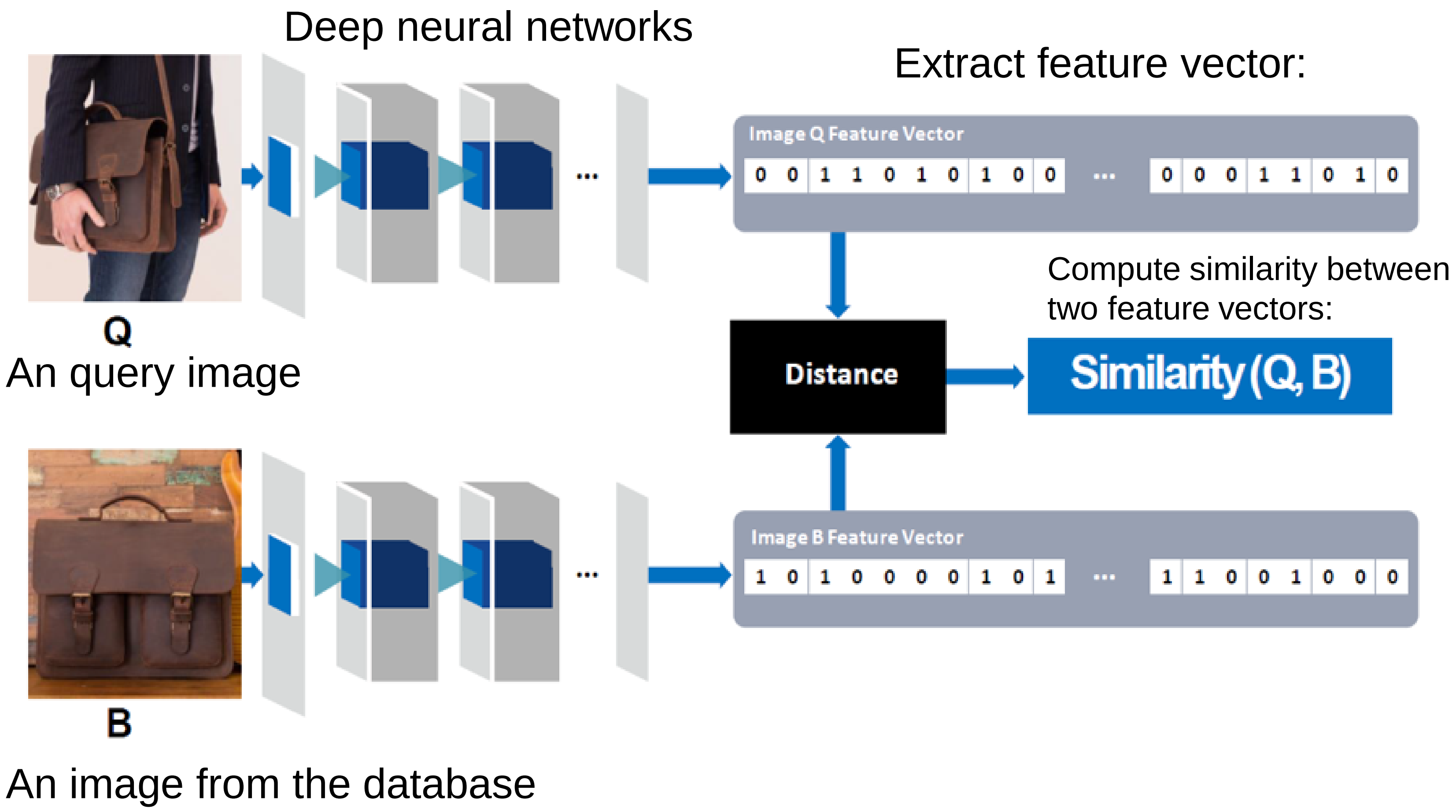
Image database
(e.g, billions of images)

A simple method (linear search):

1. Compute similarity between the query image and all images in the database.

2. Return the top-k similar images in the database.

3. Computational expensive for large scale database (e.g., 1 billion images in the database)



Similarity calculation for two images

https://web.stanford.edu/class/cs246/

Nearest Neighbour Search (NNS)

- k-nearest neighbour search
  - Given a query, identify the top-k nearest neighbours (top-k most similar items) in the database
  - The result is based on a certain type of similarity metric. (e.g., L2 distance)

- Similarity metrics
  - Euclidean distance (L2 distance)
  - Cosine similarity
  - Hamming distance (for binary data)
  - Manhattan distance (L1 distance)
  - Inner product
  - …

Similarity

1. Euclidean distance (L2 distance) :

$$d_{L2}(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

2. Inner product (dot product):

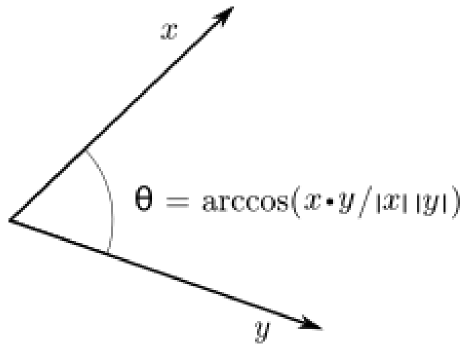
The dot product of two vectors **a** = [a_1, a_2, …, a_n] and **b** = [b_1, b_2, …, b_n] is defined as:[3]

$$\mathbf{a}\cdot\mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Geometric definition:

$$\mathbf{a}\cdot\mathbf{b} = \|\mathbf{a}\|\ \|\mathbf{b}\|\cos\theta,$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.



Similarity

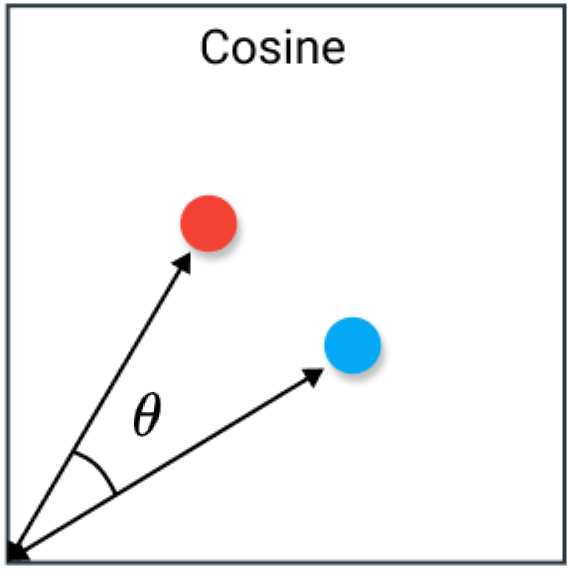
3. Cosine similarity

Given two vectors of attributes, *A* and *B*, the cosine similarity, $\cos(\theta)$, is represented using a dot product and magnitude as

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}},$$

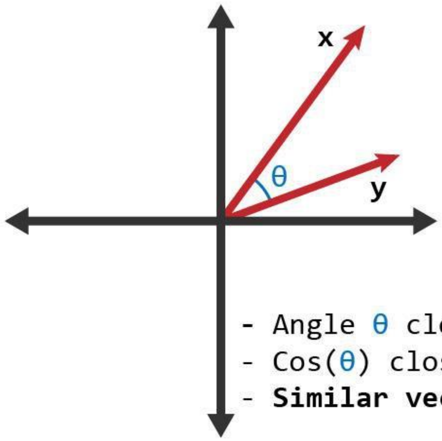
where A_i and B_i are components of vector A and B respectively.

Cosine Distance = 1−Cosine Similarity

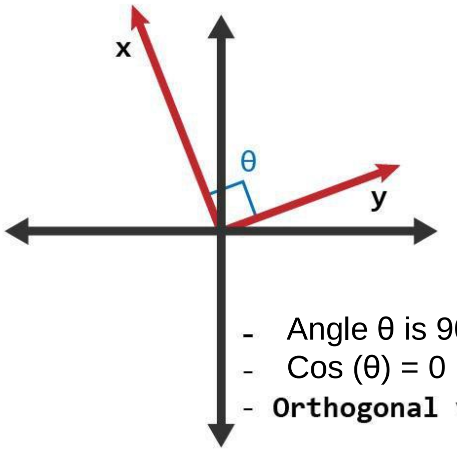


Similarity

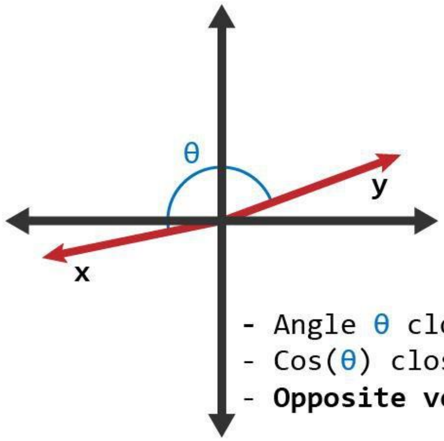
$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$



- Angle θ close to 0
- Cos(θ) close to 1
- **Similar vectors**



- Angle θ is 90
- Cos (θ) = 0
- **Orthogonal vectors**



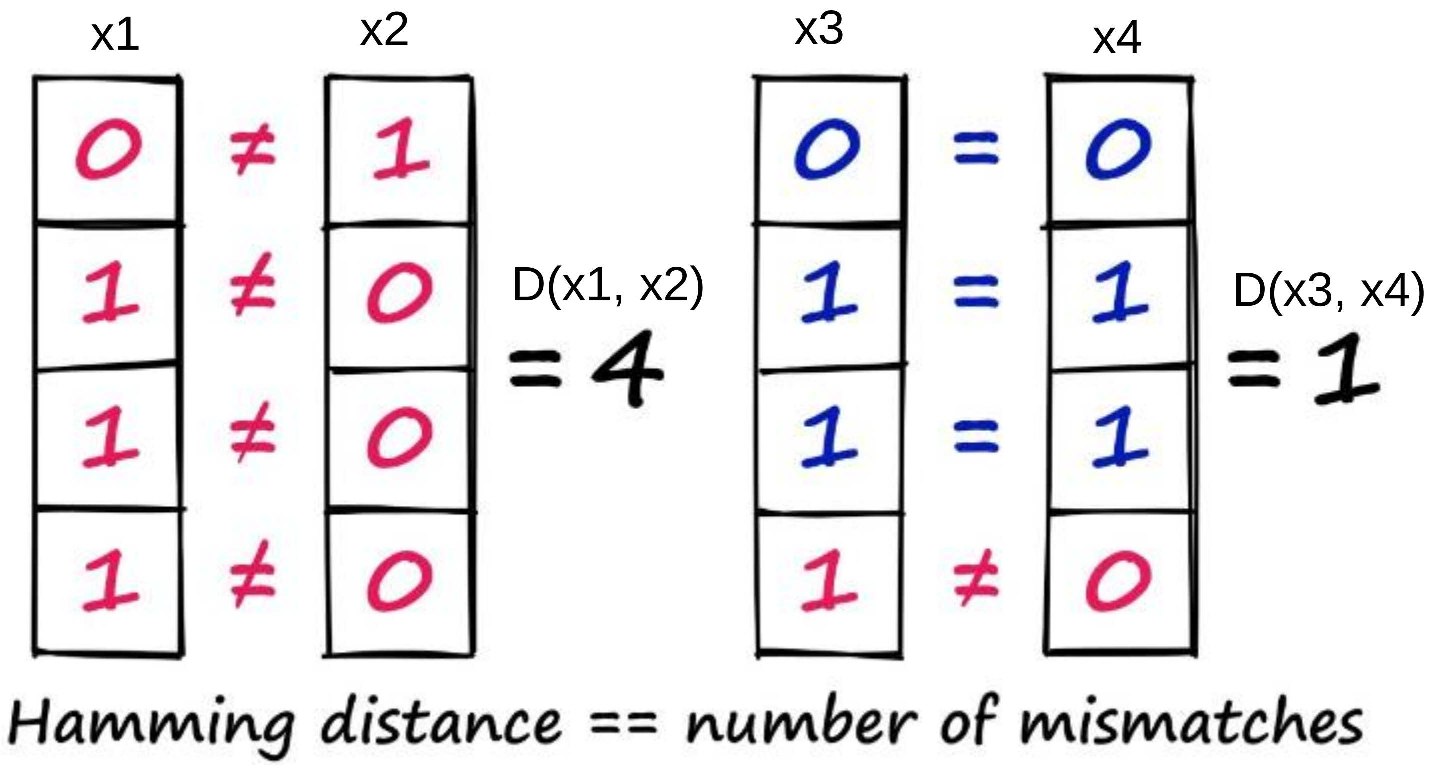
- Angle θ close to 180
- Cos(θ) close to -1
- **Opposite vectors**

https://www.learndatasci.com/glossary/cosine-similarity/

Similarity

4. Hamming distance:

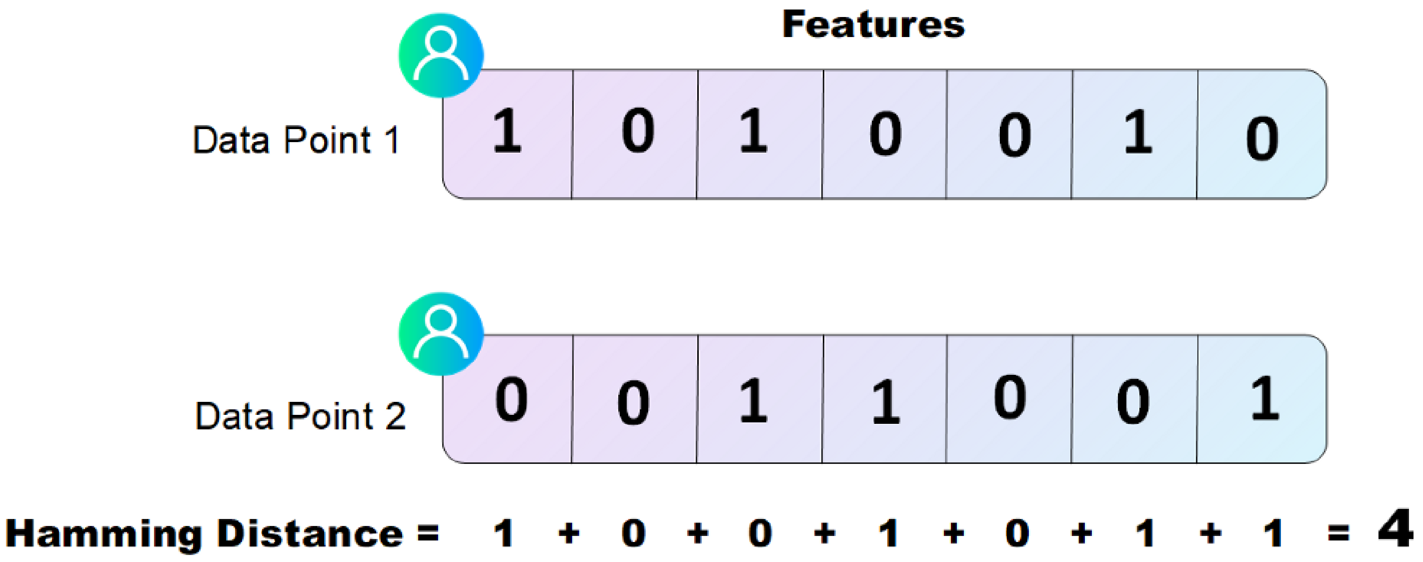
- The number of positions with different values.
- Binary data: the number of different bits in two binary vectors.
- A good option for similarity calculation of binary data



https://www.pinecone.io/learn/locality-sensitive-hashing-random-projection/

Similarity

Hamming distance on binary vectors:



https://aigents.co/data-science-blog/publication/distance-metrics-for-machine-learning

Nearest Neighbour Search (NNS)

- Nearest Neighbour Search (NNS)
  - Linear search
    - Also called linear scan, exhaustive search
    - Produce extract search results
    - Slow
  - Approximate Nearest Neighbour Search (ANN)
    - Return approximate search results
      - Local Sensitive Hashing (LSH)
      - Product Quantization (PQ)
      - Inverted File Index (IVF)
      - ….
    - A more comprehensive list:
      - https://github.com/erikbern/ann-benchmarks

Linear search

- Linear search (exhaustive search)
  - Given a query vector, compute the similarity with all samples in the database, return the top-k most similar samples

Example:
Question: given the query sample and the samples in the database below, find the top 2 nearest neighbours in the database.

Query sample:

A	[0, -1]

Database samples (5):

B	[-2, 0]
C	[1, 2]
D	[2, 1]
E	[1, -1]
F	[-1, 2]

Linear search

Example:
Question: given the query sample and the samples in the database below, find the top 2 nearest neighbours in the database.

Query sample:

A	[0, -1]

Database samples (5):

B	[-2, 0]
C	[1, 2]
D	[2, 1]
E	[1, -1]
F	[-1, 2]

Squared L2 distance:

$$\|x - y\| = \sum_{i=1}^{d} (x_i - y_i)^2$$

	B (-2, 0)	C (1, 2)	D (2, 1)	E (1, -1)	F (-1, 2)
A (0, -1)	5	10	8	1	10

Search result: the top 2 similar items are E and B
We need 5 distance calculations (we have 5 items in the database)

Linear search

- Linear search (exhaustive search)
  - Given a query vector, compute the similarity with all samples in the database, return the top-k most similar samples
    - For one query, search complexity: O(n),
      n: the number of samples in the database
  - Not efficient for large databases and high dimensional data
    - Imagine a dataset containing millions or even billions of samples
    - Generally, one similarity calculation for high dimensional data is expensive

Outline

- Nearest Neighbour Search (NNS) Problems
- **Local Sensitive Hashing (LSH) – Random Projection**
- Product Quantization (PQ)
- Inverted File Index **(non-examinable!!)**

LSH

- Local Sensitive Hashing – Random Projection
  - LSH hash functions
  - LSH + linear search for top-k retrieval
  - LSH + hash tables for top-k retrieval
  - Extended discussion
    - Geometry view of LSH **(non-examinable!)**

Locality Sensitive Hashing (LSH)

- LSH is a family of hashing methods
  - LSH-Random Projection
    - Random hyperplane hashing functions
    - Preserve cosine similarity
  - Many other LSH methods
    - use different types of hashing function
    - preserve different types of similarity
    - E.g., LSH-MinHashing

Many materials are from:
https://www.pinecone.io/learn/locality-sensitive-hashing-random-projection/

LSH-Random Projection

- LSH Random projection method
  - Generate K hashing functions.
    - Randomly generate K vectors
      - (using standard gaussian distribution)
    - Each vector constructs one linear hash function.
  - One hashing function transform the input vector into one binary value (1-bit)
  - K hashing functions transform the input vector into a K-bit binary vector (hash vector, binary code).

One hashing function:
$$h(x) = \begin{cases} 1 & : w^{\mathrm{T}}x > 0 \\ 0 & : w^{\mathrm{T}}x \leq 0 \end{cases}$$

x : the input vector

w : the parameters for the hashing function (the randomly generated vector)

LSH-Random Projection

Example (LSH + linear search):

Question: given the query sample and the samples in the database below, find the top 2 nearest neighbours in the database. Use LSH-random projection.

Query sample:

A	[0, -1]

Database samples (5):

B	[-2, 0]
C	[1, 2]
D	[2, 1]
E	[1, -1]
F	[-1, 2]

4 hashing functions are given:

w_1 =[-1 1];

w_2 =[-1 0];

w_3 =[0 1];

w_4 =[1 -1];

LSH-Random Projection

Solution:

Step1: convert the database items and the query item into binary vectors.

Query sample:

A	[0, -1]

Database samples (5):

B	[-2, 0]
C	[1, 2]
D	[2, 1]
E	[1, -1]
F	[-1, 2]

4 hashing functions are given:

$w_1 = [-1\ 1]^T$;
$w_2 = [-1\ 0]^T$;
$w_3 = [\ 0\ 1]^T$;
$w_4 = [1\ -1]^T$;

$h_1 :\ w_1^T x_A = -1 * 0 + 1 * (-1) = -1 <= 0 \ --> h_1(x_A) = 0$
$h_2 :\ w_2^T x_A = -1 * 0 + 0 * (-1) = 0 <= 0 \ --> h_2(x_A) = 0$
$h_3 :\ w_3^T x_A = 0 * 0 + 1 * (-1) = -1 <= 0 \ --> h_3(x_A) = 0$
$h_4 :\ w_4^T x_A = 1 * 0 + (-1) * (-1) = 1 > 0 \ --> h_4(x_A) = 1$

The binary vector for the query sample A is [0 0 0 1]

LSH-Random Projection

4 hashing functions are given:
$w_1 = [-1\ 1]^T$;
$w_2 = [-1\ 0]^T$;
$w_3 = [\ 0\ 1]^T$;
$w_4 = [1\ -1]^T$;

Database samples (5):

B	[-2, 0]
C	[1, 2]
D	[2, 1]
E	[1, -1]
F	[-1, 2]

$h_1 :\ w_1^T x_B = -1 * (-2) + 1 * 0 = 2 > 0 \dashrightarrow h_1(x_B) = 1$
$h_2 :\ w_2^T x_B = -1 * (-2) + 0 * 0 = 2 > 0 \dashrightarrow h_2(x_B) = 1$
$h_3 :\ w_3^T x_B = 0 * (-2) + 1 * 0 = 0 <= 0 \dashrightarrow h_3(x_B) = 0$
$h_4 :\ w_4^T x_B = 1 * (-2) + (-1) * 0 = -2 <= 0 \dashrightarrow h_4(x_B) = 0$

The binary vector for the database sample B is [1 1 0 0]

LSH-Random Projection

4 hashing functions are given:

$w_1 = [\ -1\ 1]^T;$
$w_2 = [\ -1\ 0]^T;$
$w_3 = [\ 0\ 1]^T;$
$w_4 = [\ 1\ -1]^T;$

Database samples (5):

B	[-2, 0]
C	[1, 2]
D	[2, 1]
E	[1, -1]
F	[-1, 2]

$w_1^T x_C = -1 * 1 + 1 * 2 = 1 > 0 \dashrightarrow h_1(x_C) = 1$
$w_2^T x_C = -1 * 1 + 0 * 2 = -1 <= 0 \dashrightarrow h_2(x_C) = 0$
$w_3^T x_C = 0 * 1 + 1 * 2 = 2 > 0 \dashrightarrow h_3(x_C) = 1$
$w_4^T x_C = 1 * 1 + (-1) * 2 = -1 <= 0 \dashrightarrow h_4(x_C) = 0$

The binary vector for the database sample C is [1 0 1 0]

$w_1^T x_D = -1 * 2 + 1 * 1 = -1 <= 0 \dashrightarrow h_1(x_D) = 0$
$w_2^T x_D = -1 * 2 + 0 * 1 = -2 <= 0 \dashrightarrow h_2(x_D) = 0$
$w_3^T x_D = 0 * 2 + 1 * 1 = 1 > 0 \dashrightarrow h_3(x_D) = 1$
$w_4^T x_D = 1 * 2 + (-1) * 1 = 1 > 0 \dashrightarrow h_4(x_D) = 1$

The binary vector for the database sample D is [0 0 1 1]

LSH-Random Projection

4 hashing functions are given:

$w_1 = [-1\ 1]^T$;
$w_2 = [-1\ 0]^T$;
$w_3 = [\ 0\ 1]^T$;
$w_4 = [1\ -1]^T$;

Database samples (5):

B	[-2, 0]
C	[1, 2]
D	[2, 1]
E	[1, -1]
F	[-1, 2]

$w_1^T x_E = -1 * 1 + 1 * (-1) = -2 \ \ <=0 \ \ --> h_1(x_E) = 0$
$w_2^T x_E = -1 * 1 + 0 * (-1) = -1 \ \ <=0 \ \ --> h_2(x_E) = 0$
$w_3^T x_E = \ 0 * 1 + 1 * (-1) = -1 \ \ <=0 \ \ --> h_3(x_E) = 0$
$w_4^T x_E = \ 1 * 1 + \ (-1) * (-1) = 2 > 0 --> h_4(x_E) = 1$

The binary vector for the database sample E is [0 0 0 1]

$w_1^T x_F = -1 * (-1) + 1 * 2 = 3 \ \ > 0 \ \ --> h_1(x_F) = 1$
$w_2^T x_F = -1 * (-1) + 0 * 2 = 1 \ \ > 0 \ \ --> h_2(x_F) = 1$
$w_3^T x_F = \ 0 * (-1) + 1 * 2 = 2 \ \ > 0 \ \ --> h_3(x_F) = 1$
$w_4^T x_F = \ 1 * (-1) + \ (-1) * 2 = -3 <= 0 --> h_4(x_F) = 0$

The binary vector for the database sample F is [1 1 1 0]

Query sample:

A	[0, -1]

Database samples (5):

B	[-2, 0]
C	[1, 2]
D	[2, 1]
E	[1, -1]
F	[-1, 2]

$$\xrightarrow{\text{After hashing functions}}$$

Query sample:

A	[0 0 0 1]

Database samples (5):

B	[1 1 0 0]
C	[1 0 1 0]
D	[0 0 1 1]
E	[0 0 0 1]
F	[1 1 1 0]

Query sample:

A	[0 0 0 1]

Database samples (5):

B	[1 1 0 0]
C	[1 0 1 0]
D	[0 0 1 1]
E	[0 0 0 1]
F	[1 1 1 0]

Step2: perform linear search on binary vectors using hamming distance D_H

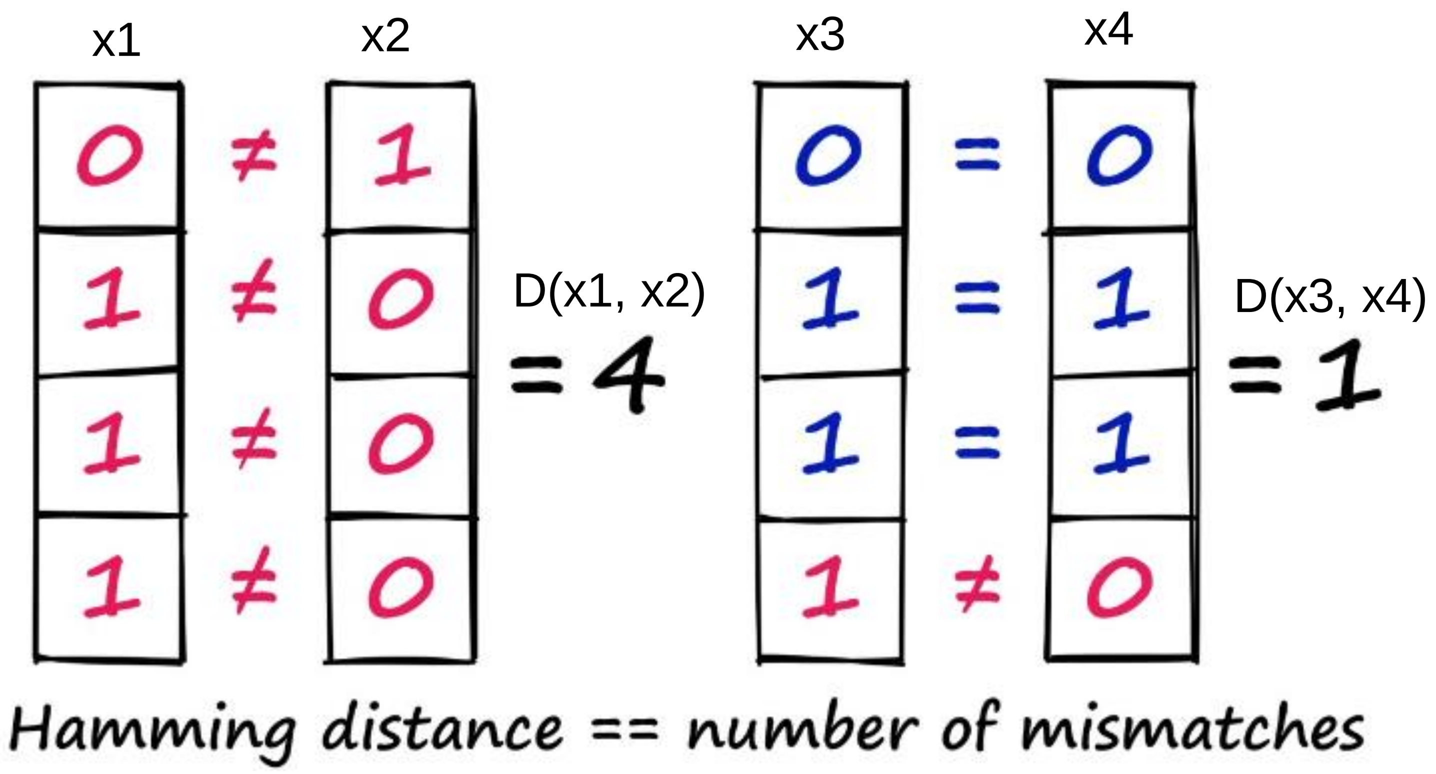
D_H (A, B) = 3 (number of mismatched bits)
D_H (A, C) = 3
D_H (A, D) = 1
D_H (A, E) = 0
D_H (A, F) = 4

Search result:
Given the query A,
the top 2 nearest neighbours are: E and D

Similarity (recap)

4. Hamming distance:

- Generally, counts the number of positions where two symbols are different.
- For binary data: the number of bits that are different in two binary vectors.



https://www.pinecone.io/learn/locality-sensitive-hashing-random-projection/

- L2 distance based linear search VS LSH + linear search

L2 distances

	B (-2, 0)	C (1, 2)	D (2, 1)	E (1, -1)	F (-1, 2)
A (0, -1)	5	10	8	1	10

Search result: the top 2 similar items are E and B

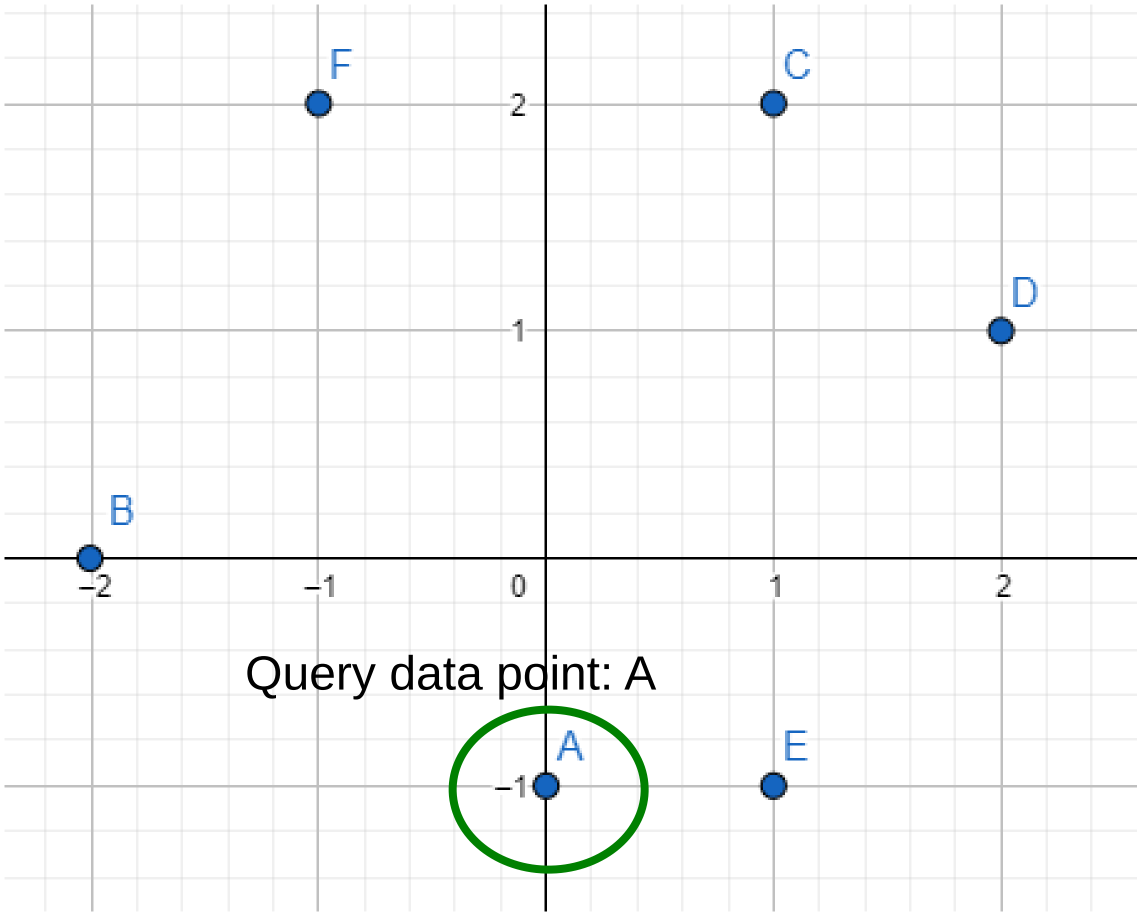
LSH and Hamming distances

	B	C	D	E	F
A	3	3	1	0	4

Search result: the top 2 similar items are E and D

Data points:

A	(0, -1)
B	(-2, 0)
C	(1, 2)
D	(2, 1)
E	(1, -1)
F	(-1, 2)



LSH and Hamming distances

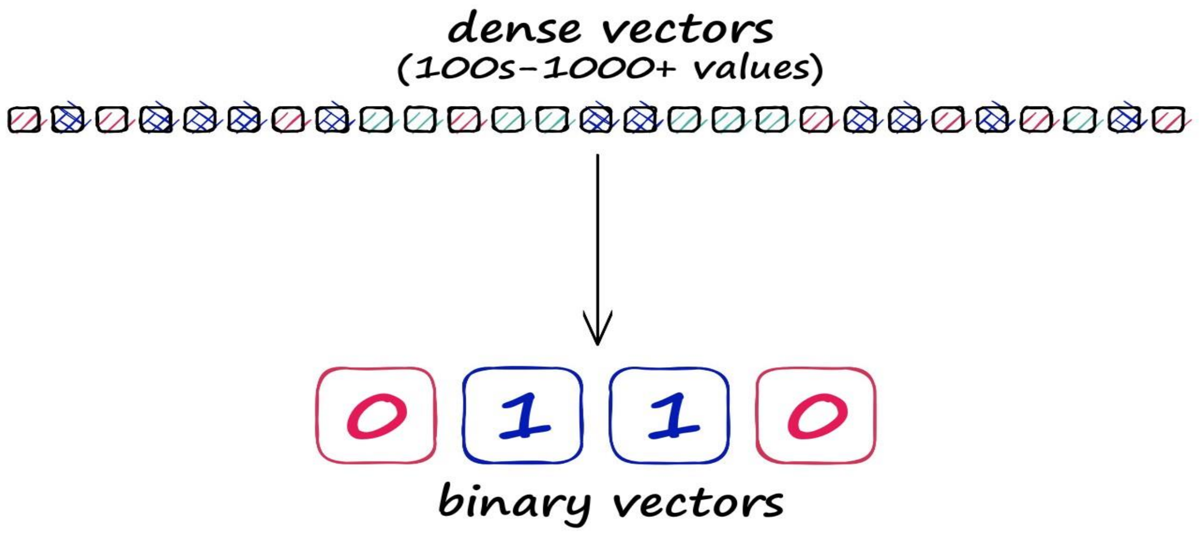
	B	C	D	E	F
A	3	3	1	0	4

L2 distances on the original features

	B	C	D	E	F
A	5	10	8	1	10

LSH for similarity search

- With LSH, we can
  - generate low-dimensional binary features from high-dimensional input
  - Preserve the similarity after binary mapping.
    - If x is similar to y in the original space, after binary mapping, their binary vectors are also similar in the binary space.



Discussion

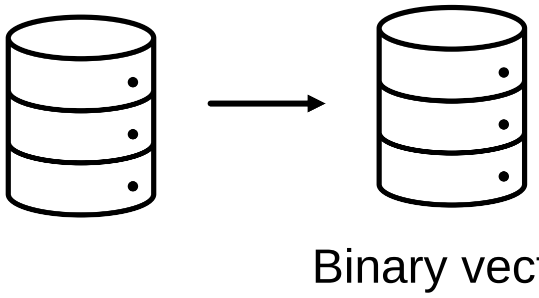
- Why LSH + linear search is more efficient than L2 distance + linear search?

Discussion

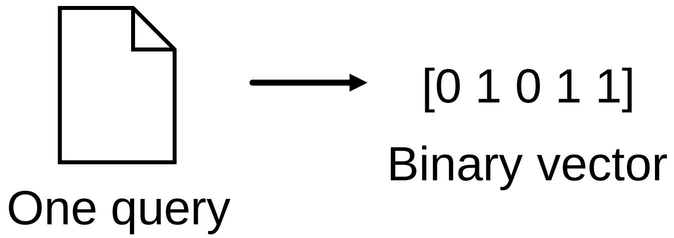
- Why LSH + linear search is more efficient than L2 distance + linear search?
  - We can use the efficient hamming distance on the binary vectors.
  - Hamming distance calculation is very efficient
    - XOR operation on the bits (counting mismatched bits)
    - Much more efficient than L2 distance on float values
    - Especially useful for high-dimensional data

- LSH requires extra calculation for binary transform
  - Database binary transform with LSH is done offline.
    - The whole database are transformed to binary vectors for only one time (offline pre-processing)
    - The database binary vectors will be used for all queries.
    - For one query, we only need to transform the query input vector to binary vector.

Step1: convert the database into binary vectors (offline step)



Step 2: process query requests
For each query, we only need to convert the query to a binary vector (online processing)



Discussion

- Benefits of using binary vectors
  - Binary vectors are efficient for storage, transmission and similarity calculation (Hamming distance).
    - A 128-dimensional float-value vector:
      128 x 4 byte = 512 bytes
      - 1 float value requires 4 bytes (32 bits)
    - A 128 bit binary vector = 16 bytes
      - 1 byte = 8 bits,

Discussion

- More hashing functions will increase the accuracy on top-k nearest neighbour search.
    - More hashing functions lead to longer binary vectors (we have more bits)
    - With more bits, the top-k ranking results of LSH will be more similar to the cosine distance or L2 distance based top-k results.

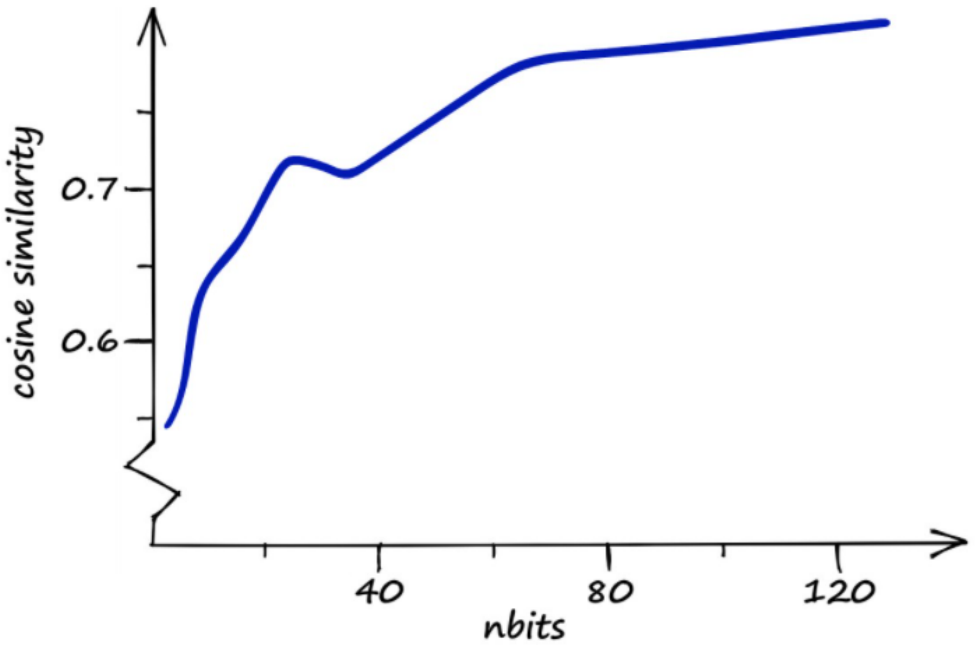
Analysis on the number of bits

Number of bits (nbits): the number of binary values in the binary vector. It is equal to the number of hash functions.

With more bits, the top-k retrieval results become more accurate.

Y axis: average cosine similarity of top-100 results.

It indicates the accuracy of the retrieved results



As we increase vector resolution with **nbits**, our results will become more precise — here, we can see that a larger **nbits** value results in higher cosine similarity in our results.

Sift1M dataset, 1M samples, 128 dimentions
Facebook Faiss implementation, top 100 retrieved
https://www.pinecone.io/learn/locality-sensitive-hashing-random-projection/

More hashing functions

- We can modify the LSH example to generate more hashing functions and observe the difference.

Query sample:

A	[0, -1]

Hashing functions:
$w_1 = [\ -1\ 1]^T$;
$w_2 = [\ -1\ 0]^T$;
$w_3 = [\ 0\ 1]^T$;
$w_4 = [\ 1\ -1]^T$;

Add more hash functions:
$w_5 = [\ 1\ 0]^T$;
$w_6 = [\ -1\ -1]^T$;

Database samples (5):

B	[-2, 0]
C	[1, 2]
D	[2, 1]
E	[1, -1]
F	[-1, 2]

We will explore this question in the tutorial class

LSH for similarity search

- LSH for similarity search

  Two different approaches

  - 1. Using linear search on the binary vectors
    - As shown in the previous example
  - 2. Using hash tables for similarity search

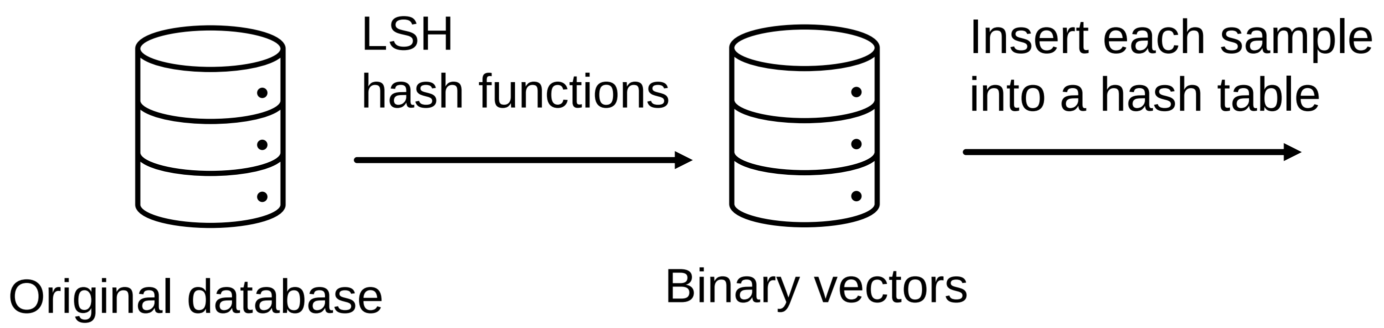
Using hash tables for LSH

- 2. Similarity search using hash tables
  - Method1: Using one hash table
  - Method2: Using multiple hash tables

Using hash tables for LSH

Method 1: using one hash table

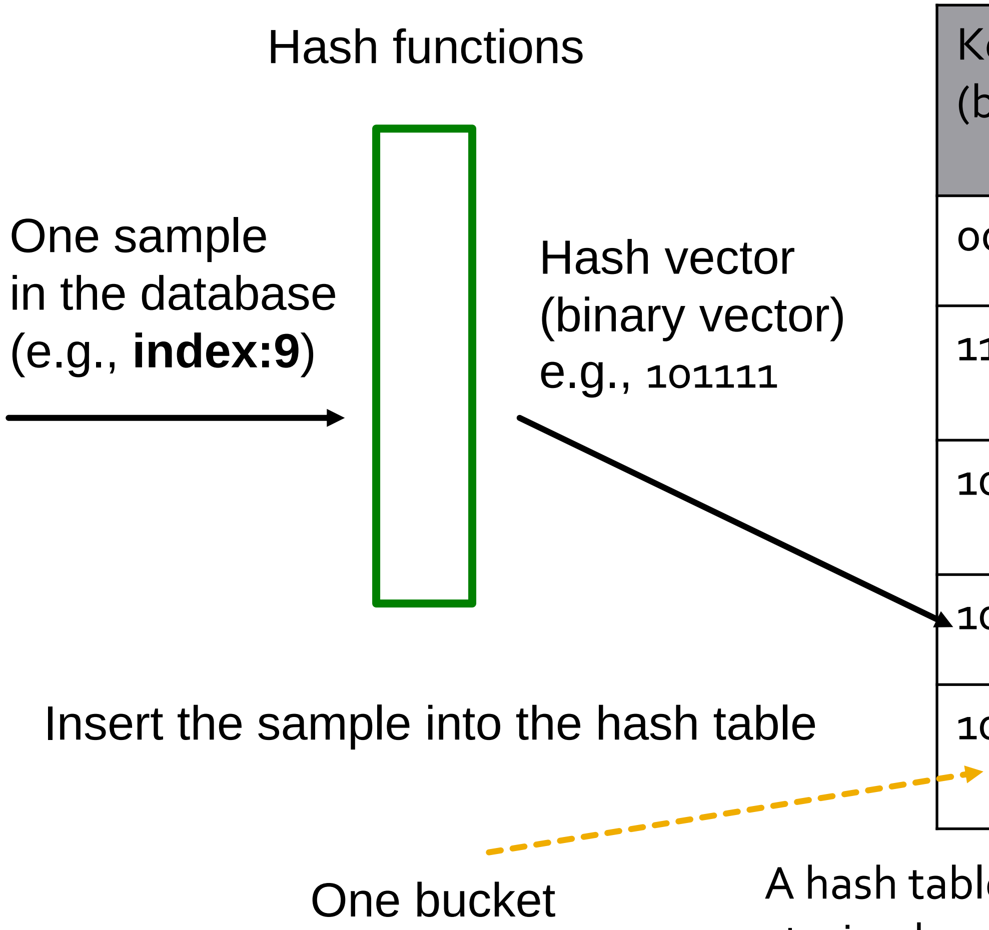
- Step 1: build the hash table for the database
  - Generate the hash vector (h) for each data sample in the database and insert <h, sample_idx> into a hash table.



0100	23,34
1000	11
0011	56,78
1010	1
0001	5,60

a hash table

Create hash table


Hash functions
One sample in the database (e.g., **index:9**)
Hash vector (binary vector) e.g., 101111
Insert the sample into the hash table
One bucket

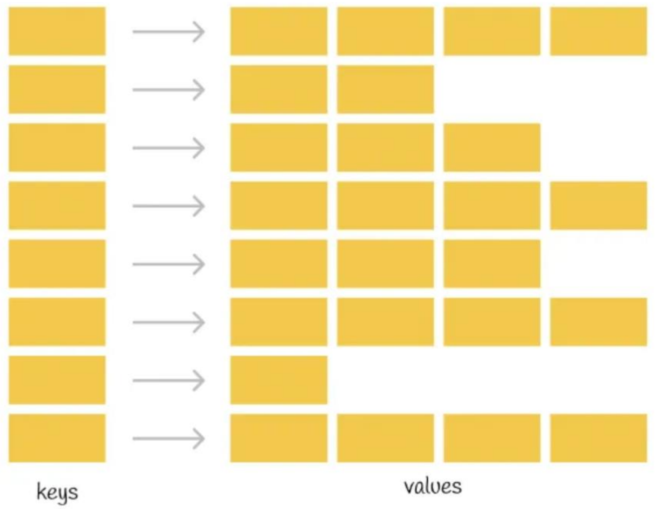
Hash table
(hash buckets)

Key (binary vector)	Value (sample index list)
001111	(1, 3, 100)
111100	(13)
100110	()
101111	(14, 18, **9**)
101010	(15, 19, 29, 39)

A hash table can be implemented by a data struct for storing key-value pairs (an associative array), e.g., the dictionary data type in Python

- Hash table, bucket and hash vector
  - A hash table is a key-value data structure
  - can be implemented by an associative array for storing key-value entries
    - e.g., the dictionary data structure in Python for storing key-value items
  - The entries in a hash table are called buckets

A hash table aims to group similar data into the same bucket: similar data are likely to have the same key (binary vector)



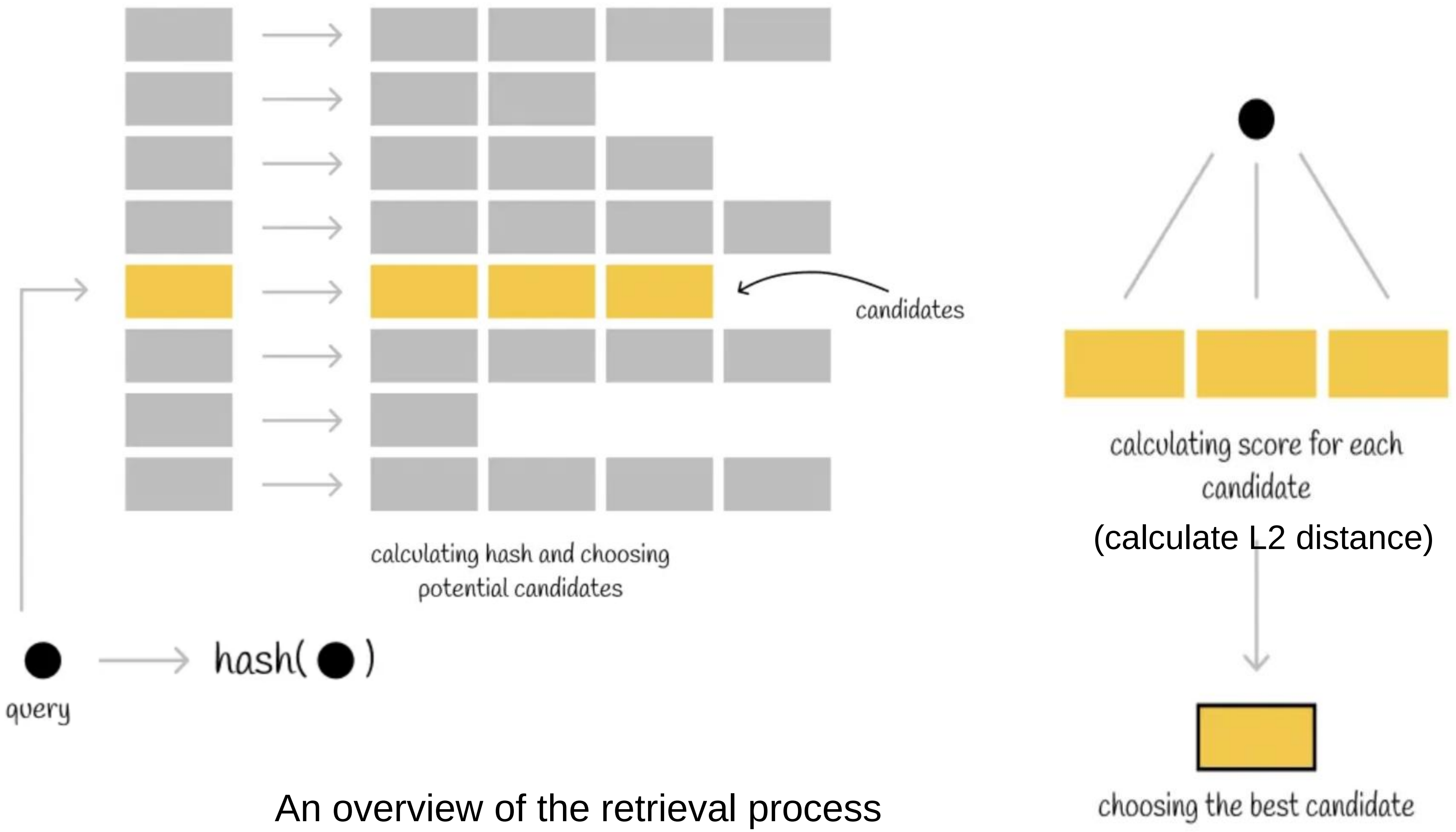
- Each entry is a key-value pair: <h, sample_idx_list>
  - h is a hash vector, sample_idx_list is a list of the indexes of the data samples in the database

- If two hash vectors (hash codes) are the same, they will be mapped to the same bucket in the hash table.
  - The entries in the same bucket will have the same hash vector (hash code).

Two examples in the database have the same hash code (binary vector) $\xleftrightarrow{\text{indicates}}$ Two examples are mapped to the same bucket in a hash table

Method 1: exhaustive search of all buckets

- Step 1: Build the hash table for the database **(done)**
- Step 2: process queries
    - Compute the binary vector for the query sample.
        - E.g input query x -> [101010]
    - Retrieved the candidate items which have the same binary vector as the query.
    - Return top-k items from the candidate set using L2 distance.


candidates
calculating hash and choosing potential candidates
hash(●)
query
calculating score for each candidate
(calculate L2 distance)
choosing the best candidate

An overview of the retrieval process

Vyacheslav Efimov, https://towardsdatascience.com/similarity-search-knn-inverted-file-index-7cab80cc0e79

Hash table example

Hash table (database)
(hash buckets)

Key (binary vector) (buckets)	Value (sample index list)
001111	(1, 3, 100)
111100	(13)
100110	()
101011	(14, 18, 9, 22, 46)
101010	(15, 19, 29, 39)

One query x -> generate the binary vector

Query: [101011]

Retrieve the bucket from the hash table

Candidate items (14, 18, 9, 22, 46))

Top-k retrieved results

Use L2 distance to get the top k items from the candidate set

Using hash tables for LSH

- More hashing functions
  - --> leads to longer binary vectors
  - --> more bits
  - --> more buckets
    - #bits: 2: maximum #buckets: 2^2= 4
    - #bits: 4, maximum #buckets: 2^4=16
    - #bits: 8, maximum #buckets: 2^8=256
    - #bits: 16, maximum #buckets: 2^{16}=65536

Using hash tables for LSH

- Efficiency of using LSH + hash table:
(compared to linear search)
  - 1. the number of distance calculation is greatly reduced
    - Only calculate the L2 distance in the candidate set (the retrieved bucket).
    - Do not need to perform a linear scan to calculate distance for every item in the database.
  - 2. The binary vector generation is efficient
    - In the query stage, only need to map the input query into a binary vector by applying hash functions

Using hash tables for LSH

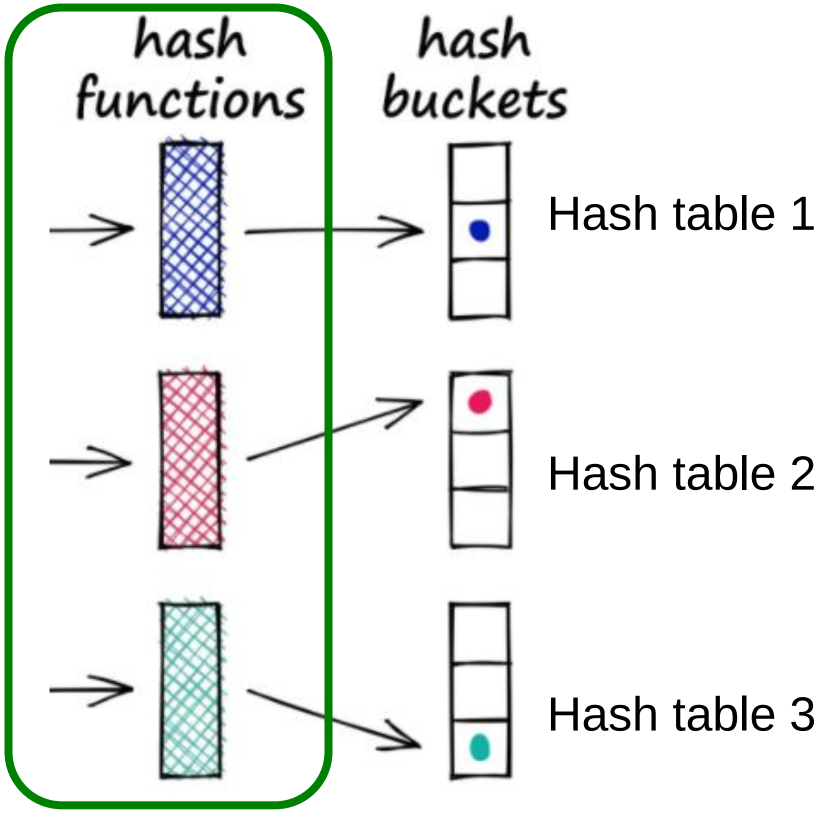
- **Method 2: using multiple hash tables**

1. Construct multiple hash tables

2. Different tables use different sets of hash functions.

Generate M different sets of LSH hash functions. One set of hash functions is for one hash table.

Example: using 3 hash tables.



For one hash table, we use its own set of LSH hash functions to generate hash vectors.

Using hash tables for LSH

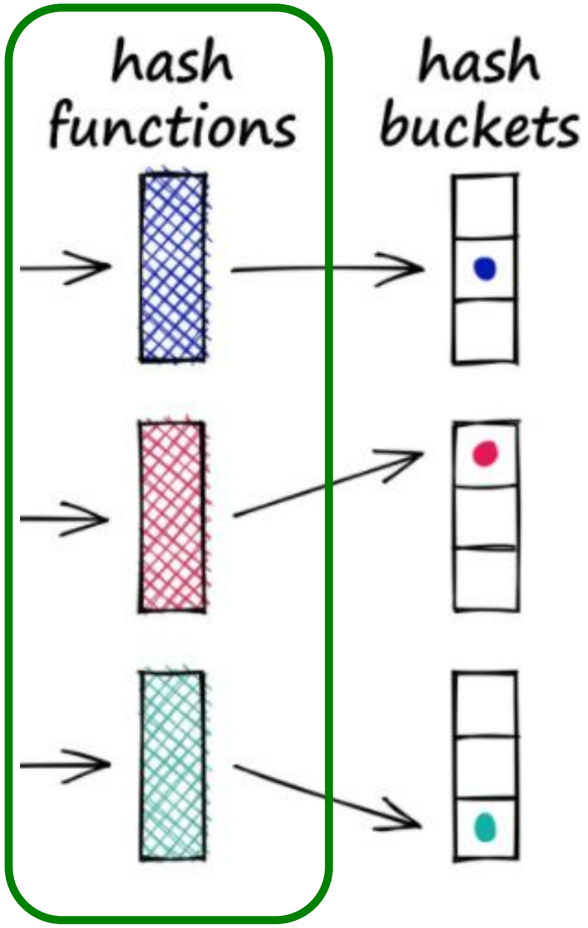
- **Method 2: using multiple hash tables (cont.)**

Query process:

1. Use the query hash vector to retrieve one bucket in each hash table.

2. Combine the retrieved results from all hash tables as candidate items

3. Perform accurate linear search on the candidate items to output top-k results



Example: using 3 hash tables.

Using hash tables for LSH

Example: use 3 hash tables. We use different hash functions for each table.

Hash table 1

Key	Value (sample_idxes)
001111	(1, 3, 100)
111100	(13, 2, 5)
100110	(15, 19, 29)
101111	(14, 18, 9)
101010	()

Query X-> [111100]

Hash table 2

Key	Value (sample_idxes)
001111	(100)
111100	()
100110	()
101111	(9, 10)
101010	(40)

Query X-> [101010]

Hash table 3

Key	Value (sample_idxes)
001111	(11, 23)
111100	(9, 21)
100110	()
101111	(30)
101010	(20)

Query X-> [101010]

1. For one query X, we will generate 3 hash vectors.
2. Retrieved one bucket from each hash table
3. Combined the retrieved buckets from each table: candidate set: {13, 2, 5, 40}
4. Linear search on the candidate set using L2 distance to return the top-k results

Extended discussion

- Extended discussion (non-examinable!)
  - Geometry view of LSH-random projection

Non-examinable!

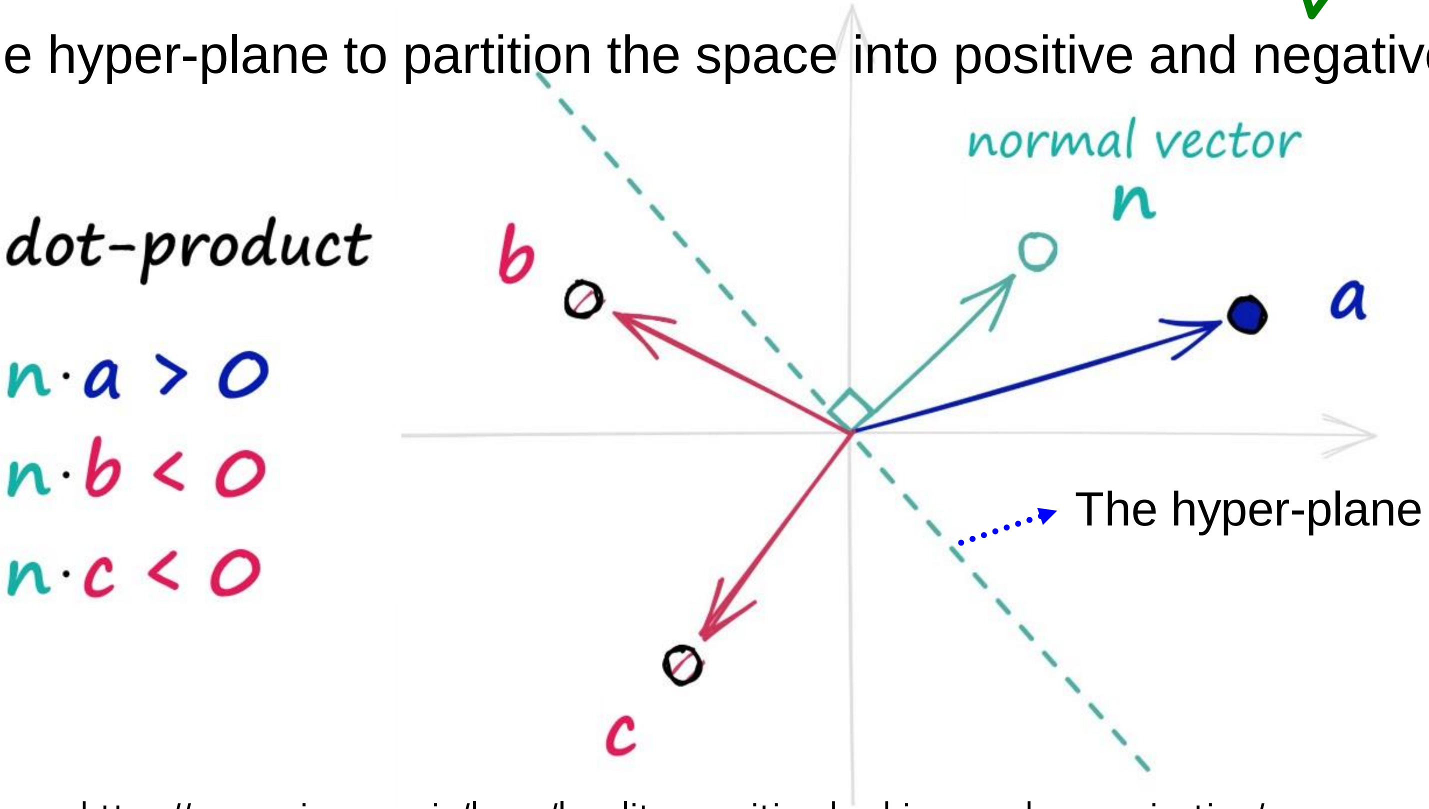
Geometry view of LSH

n: a randomly generated vector, represents one hashing function

1. Given the vector **n**, we can find a hyper-plane that
   a) passes through the origin and,
   b) the vector **n** is orthogonal to the plane
   (**n** is a normal vector to the hyper-plane).

Non-examinable!

2. The hyper-plane to partition the space into positive and negative regions



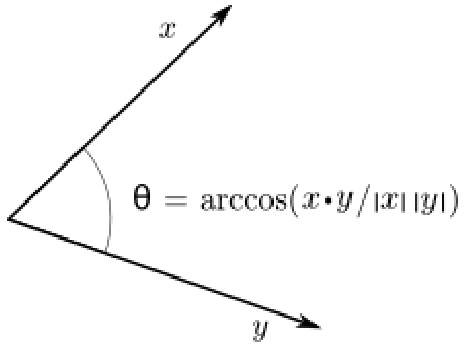
https://www.pinecone.io/learn/locality-sensitive-hashing-random-projection/

Similarity (recap)

Inner product (dot product):

The dot product of two vectors **a** = [a_1, a_2, …, a_n] and **b** = [b_1, b_2, …, b_n] is defined as:[3]

$$\mathbf{a}\cdot\mathbf{b}=\sum_{i=1}^{n}a_ib_i=a_1b_1+a_2b_2+\cdots+a_nb_n$$

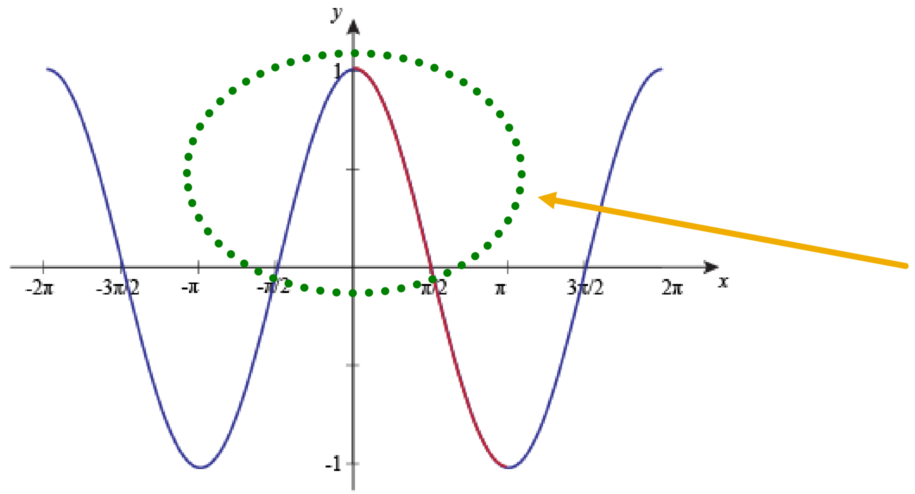


Geometric definition:

$$\mathbf{a}\cdot\mathbf{b}=\|\mathbf{a}\|\ \|\mathbf{b}\|\cos\theta,$$

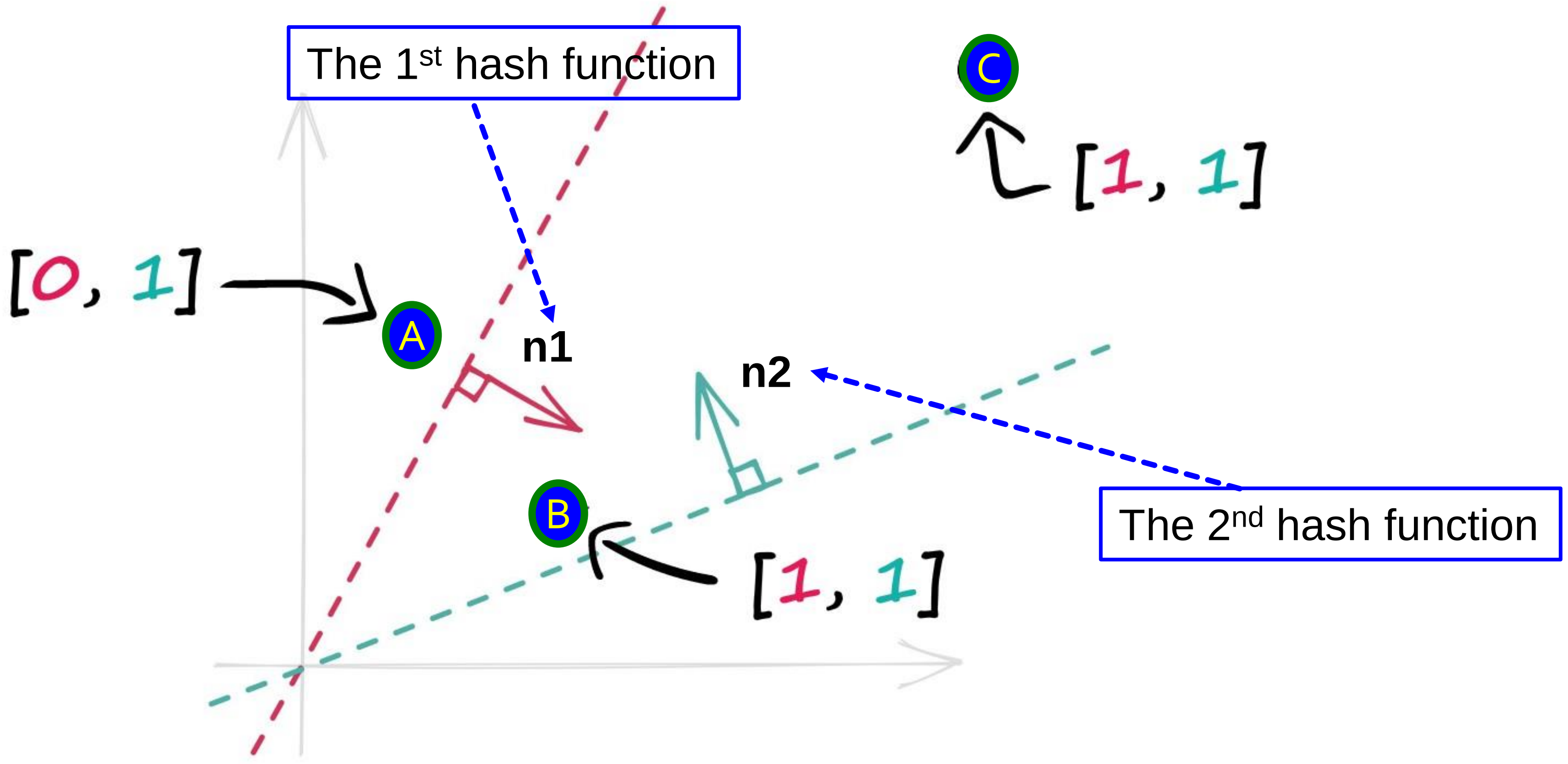
where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

$$-90^o<\theta<90^o\rightarrow\cos(\theta)>0$$



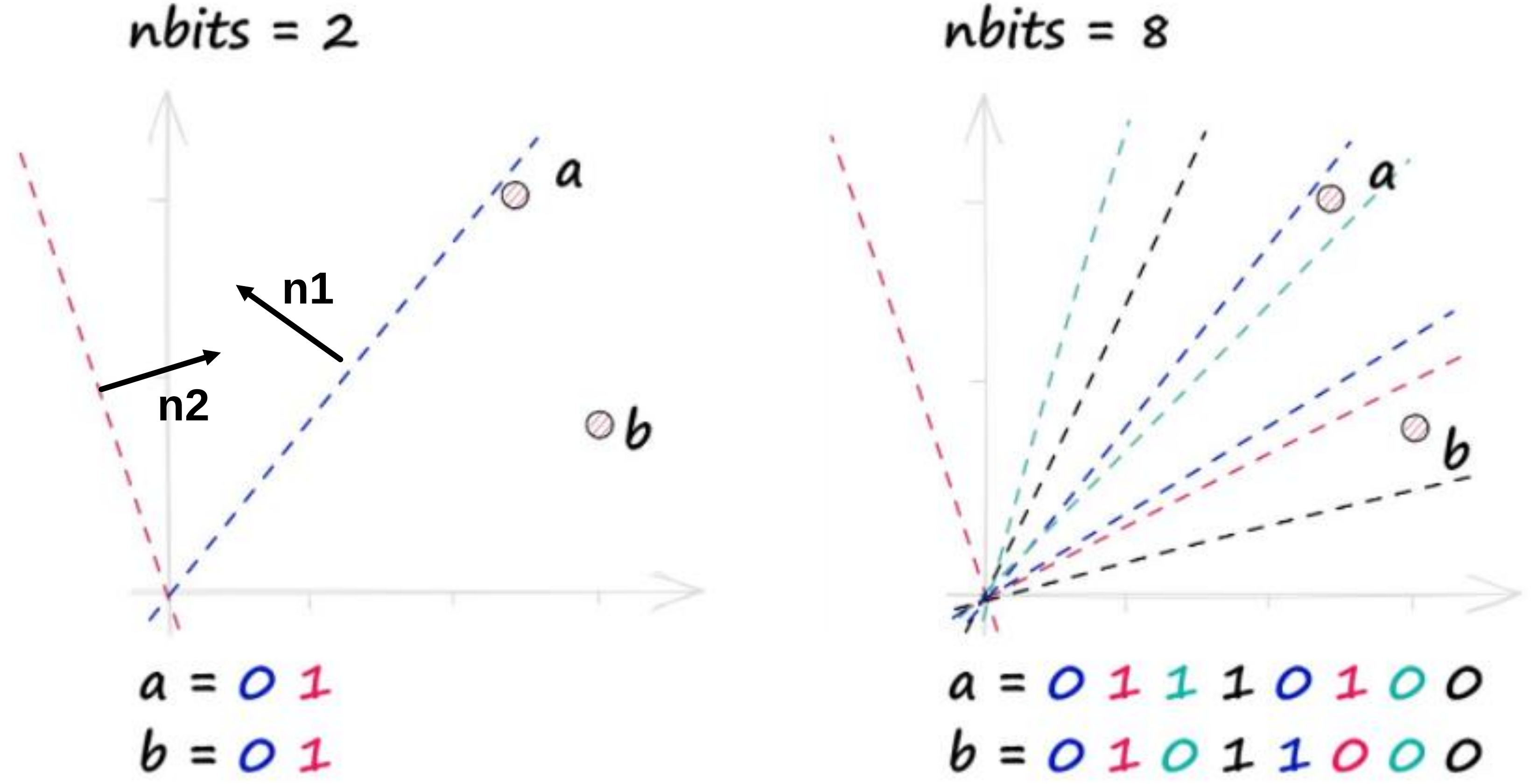
The output of the hashing function is determined by cos(θ) :

$$h(x)=\begin{cases}1 & : w^{\mathrm{T}}x>0\\ 0 & : w^{\mathrm{T}}x\le 0\end{cases}$$



One hash function (1 bit) provides weak similarity information:

It indicates whether the data points lies in the same side of the hyper-plane or not.
If they lie in the same side -> they are similar.
Otherwise, they are not similar.



Increasing the **nbits** parameter increases the number of hyperplanes used to build the binary vector representations.

Online resources

- FAISS
  - Faiss is a library for efficient similarity search and clustering of dense vectors.
  - https://github.com/facebookresearch/faiss/wiki