



Search Medium

Write



# Similarity Search, Part 7: LSH Compositions

Dive into combinations of LSH functions to guarantee a more reliable search

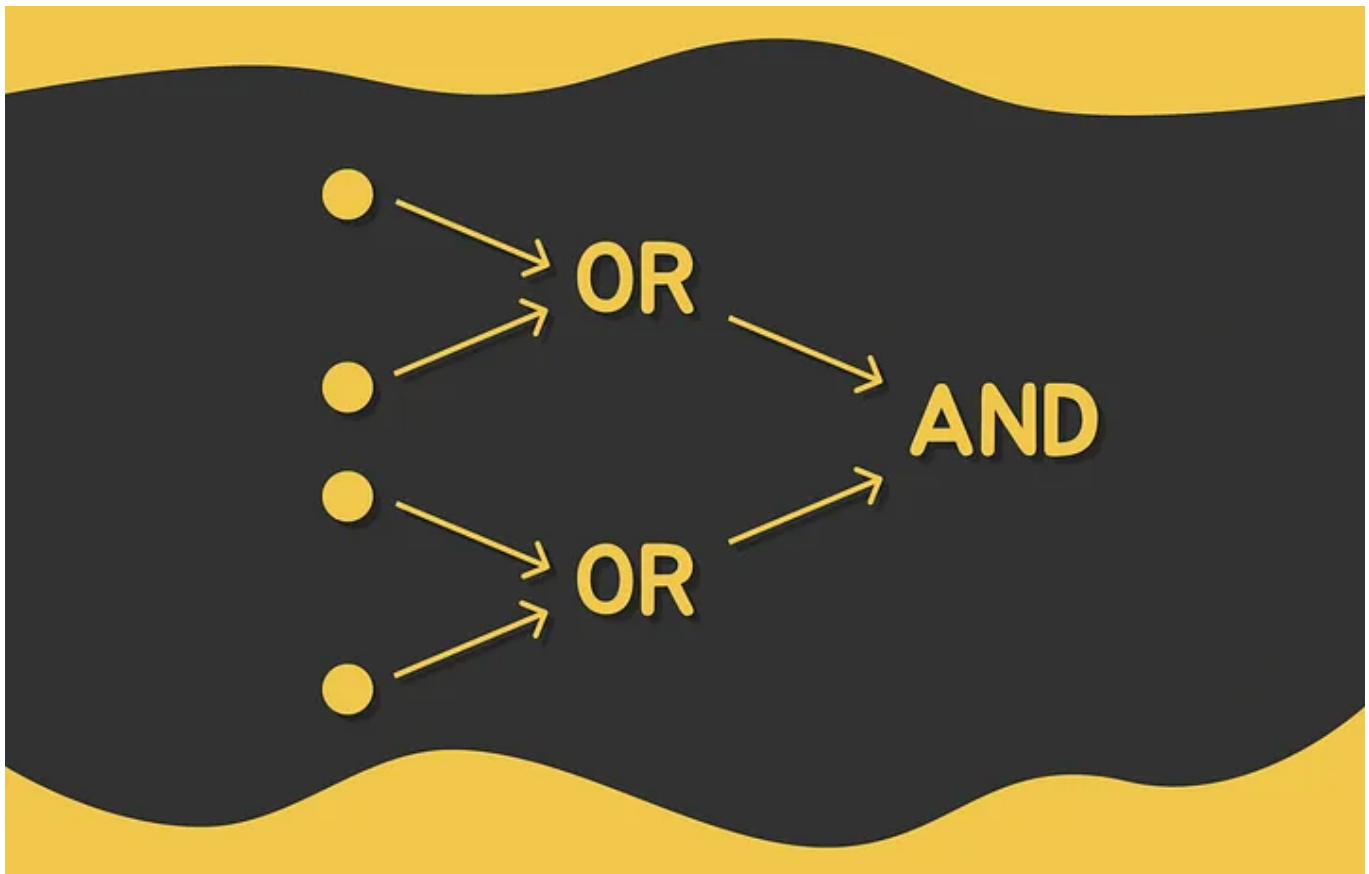


Vyacheslav Efimov · Following

Published in Towards Data Science · 11 min read · Jul 24



43



**S**imilarity search is a problem where given a query the goal is to find the most similar documents to it among all the database documents.

## Introduction

In data science, similarity search often appears in the NLP domain, search engines or recommender systems where the most relevant documents or items need to be retrieved for a query. There exists a large variety of different ways to improve search performance in massive volumes of data.

In the last two parts of this article series, we dugged into LSH — an algorithm *transforming input vectors to lower-dimensional hash values while preserving information about their similarity*. In particular, we have already looked at two algorithms that were suitable for different distance metrics:

### Similarity Search, Part 5: Locality Sensitive Hashing (LSH)

Explore how similarity information can be incorporated into hash function

[towardsdatascience.com](https://towardsdatascience.com)

Classic LSH algorithm constructs signatures that reflect the information about **Jaccard index** of vectors.

### Similarity Search, Part 6: Random Projections with LSH Forest

Understand how to hash data and reflect its similarity by constructing random hyperplanes

[towardsdatascience.com](https://towardsdatascience.com)

The method of random projections builds a forest of hyperplanes preserving **cosine similarity** of vectors.

In fact, LSH algorithms exist for other distance metrics as well. Though every method has its own unique parts, there are a lot of common concepts and formulas which appear in each of them. To facilitate the learning process of new methods in the future, we are going to focus more on the theory and provide several essential definitions and theorems which often appear in advanced LSH literature. By the end of the article, we will be able to construct more complex LSH schemes by simply combining the basic ones as LEGO building blocks.

As a bonus, at the end, we will have a look at how **Euclidean distance** can be incorporated into LSH.

*Note.* As the main prerequisite, it is expected that you are already familiar with parts 5 and 6 of this article series. If not, it is highly recommended to read them first.

*Note.* Cosine distance is formally defined in the range  $[0, 2]$ . For simplicity, we will map it to the interval  $[0, 1]$  where 0 and 1 indicate the lowest and the highest possible similarity respectively.

## Formal LSH definition

Given a distance metric  $d$ ,  $H$  is called a  $(d_1, d_2, p_1, p_2)$ -sensitive LSH function if for randomly chosen objects  $x$  and  $y$ , the following conditions are satisfied:

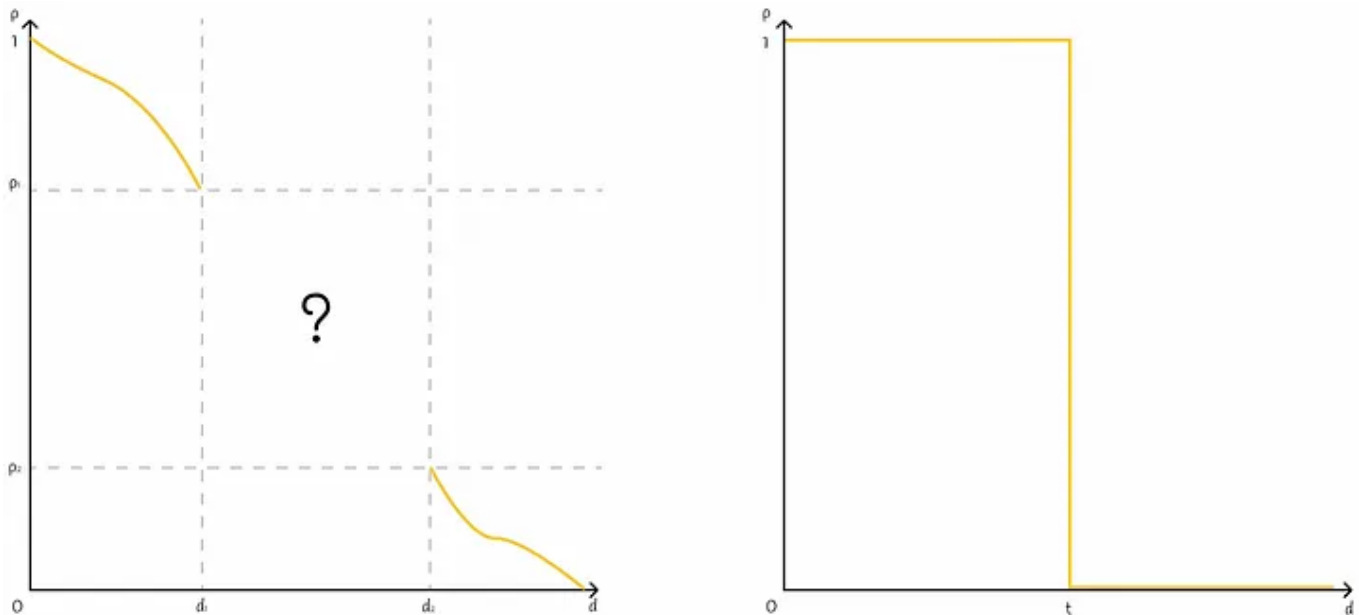
- If  $d(x, y) \leq d_1$ , then  $p(H(x) = H(y)) \geq p_1$ , i.e.  $H(x) = H(y)$  with probability at least  $p_1$ .

- If  $d(x, y) \geq d_2$ , then  $p(H(x) = H(y)) \leq p_2$ , i.e.  $H(x) = H(y)$  with probability at most  $p_2$ .

Let us understand what these statements mean. When two vectors are similar, they have a low distance between them. Basically, the first statement makes sure the probability of hashing them to the same bucket is above a certain threshold. This way, some *false negatives* are eliminated: if the distance between two vectors is larger than  $d_1$ , then the probability of them hashing to the same bucket is always less than  $p_1$ . Inversely, the second statement controls *false positives*: if two vectors are not similar and the distance between them is larger than  $d_2$ , then they have an upper probability  $p_2$  threshold of appearing in the same bucket.

Given the statement above, we normally want the following statements in the system to be satisfied:

- $p_1$  should be as close to 1 as possible to reduce the number of *false negatives*.
- $p_2$  should be as close to 0 as possible to reduce the number of *false positives*.
- The gap between  $d_1$  and  $d_2$  should be as low as possible to reduce the interval where probabilistic estimations on data cannot be made.

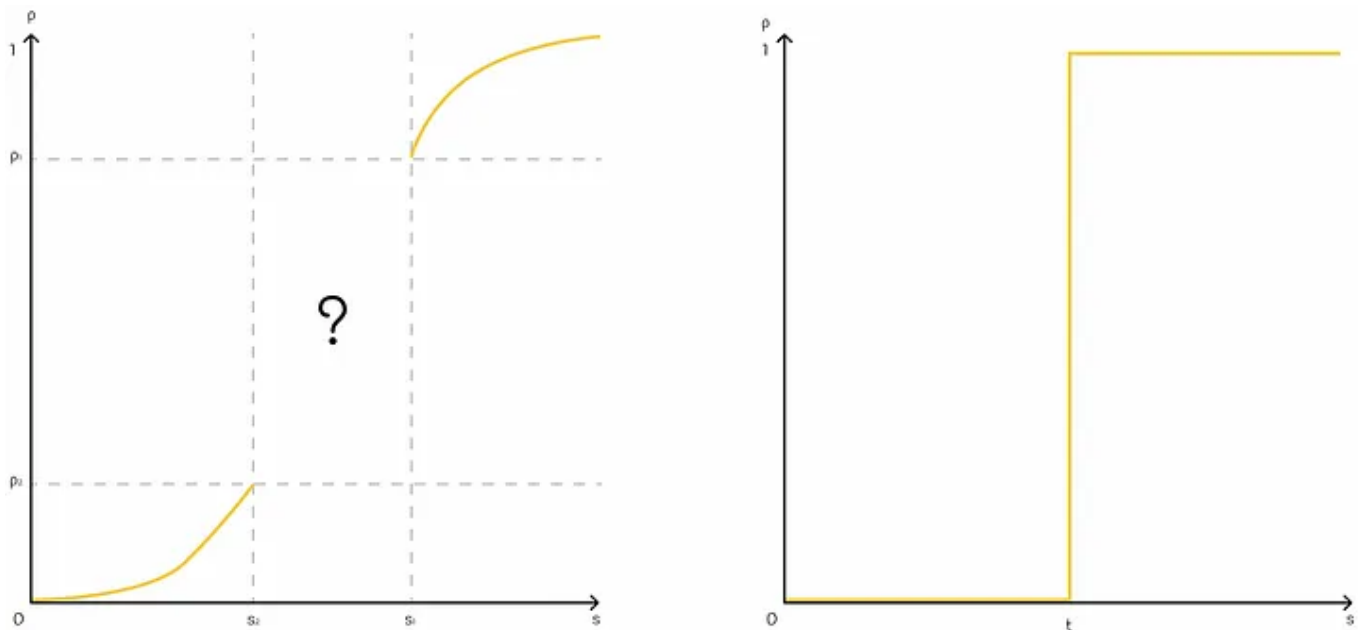


The diagram on the left shows a typical curve demonstrating relationships between LSH parameters for  $(d_1, d_2, p_1, p_2)$  notation. A curve on the right demonstrates an ideal scenario where there is not gap between thresholds  $d_1$  and  $d_2$ .

Sometimes the statement above is introduced by using the term of similarity  $s$  instead of distance  $d$ :

Given a similarity metric  $s$ ,  $H$  is called a  $(s_1, s_2, p_1, p_2)$ -sensitive LSH function if for randomly chosen objects  $x$  and  $y$ , the following conditions are satisfied:

- If  $s(x, y) \geq s_1$ , then  $p(H(x) = H(y)) \geq p_1$ , i.e.  $H(x) = H(y)$  with probability at least  $p_1$ .
- If  $s(x, y) \leq s_2$ , then  $p(H(x) = H(y)) \leq p_2$ , i.e.  $H(x) = H(y)$  with probability at most  $p_2$ .



The diagram on the left shows a typical curve demonstrating relationships between LSH parameters for  $(s_1, s_2, p_1, p_2)$  notation. A curve on the right demonstrates an ideal scenario where there is not gap between thresholds  $s_1$  and  $s_2$ .

*Note.* In this article, both notations  $(d_1, d_2, p_1, p_2)$  and  $(s_1, s_2, p_1, p_2)$  will be used. Based on the letters used in notations in the text, it should be clear whether distance  $d$  or similarity  $s$  is implied. By default, the notation  $(d_1, d_2, p_1, p_2)$  is used.

## LSH example

To make things more clear, let us prove the following statement:

If the distance metric  $s$  is Jaccard index, then  $H$  is a  $(0.6, 0.6, 0.4, 0.4)$ -sensitive LSH function. Basically, the equivalent statements have to be proved:

- If  $d(x, y) \leq 0.6$ , then  $p(H(x) = H(y)) \geq 0.4$
- If  $d(x, y) \geq 0.6$ , then  $p(H(x) = H(y)) \leq 0.4$

From the 5-th part of this article series, we know that *the probability of getting equal hash values for two binary vectors equals the Jaccard similarity*. Therefore,

if two vectors are similar by at least 40%, then it is guaranteed that the probability of getting equal hash values is also at least 40%. In the meantime, a Jaccard similarity of at least 40% is equivalent to a Jaccard index at most of 60%. As a result, the first statement is proved. The analogous reflections can be done for the second statement.

This example can be generalised into the theorem:

***Theorem.** If  $d$  is Jaccard index, then  $H$  is a  $(d_1, d_2, 1 - d_1, 1 - d_2)$ -family of LSH functions.*

Similarly, based on the results obtained from part 6, it is possible to prove another theorem:

***Theorem.** If  $s$  is cosine similarity (between -1 and 1), then  $H$  is a  $(s_1, s_2, 1 - \arccos(s_1) / 180, 1 - \arccos(d_2) / 180)$ -family of LSH functions.*

## Combining LSH functions

Let us refer to useful concepts we learned in previous parts on LSH:

- Getting back to part 5 on minhashing, every vector was split into several bands each containing a set of rows. In order for a pair of vectors to be considered candidates, there had to exist **at least one** band where **all** of the vector rows were equal.
- Regarding to part 6 on random projections, two vectors were considered candidates only if there existed **at least one** tree where **all** of the random projections did not separate the initial vectors.

As we can notice, these two approaches have a similar paradigm under the hood. Both of them consider a pair of vectors as candidates only if at **least one time** out of  $n$  configurations vectors have the same hash values **all**  $k$  times. With the boolean algebra notation, it can be written like this:

$$\begin{array}{l}
 H(x) = H(y) \quad \text{IF} \quad H_1(x) = H_1(y) \quad \text{OR} \quad H_2(x) = H_2(y) \quad \text{OR} \quad \dots \quad \text{OR} \quad H_n(x) = H_n(y), \\
 \text{where} \quad H(x) = H(y) \quad \text{IF} \quad H_1(x) = H_1(y) \quad \text{AND} \quad H_2(x) = H_2(y) \quad \text{AND} \quad \dots \quad \text{AND} \quad H_n(x) = H_n(y)
 \end{array}$$

Based on this example, let us introduce logical operators *OR* and *AND* which allow aggregating a set of hash functions. Then we will estimate how they affect the output probability of two vectors being candidates and the rate of *false negative* and *false positive* errors.

## AND operator

Given  $n$  independent LSH functions  $H_1, H_2, \dots, H_n$ , the *AND* operator considers two vectors as a candidate pair only if **all** of  $n$  corresponding hash values of both vectors are equal. Otherwise, the vectors are not considered as candidates.

If hash values of two highly different vectors are aggregated by the *AND* operator, then the probability of them being candidates decreases with the increase of the number of used hash functions. Therefore, the number of *false positive* decreases.

At the same time, two similar vectors can by chance result in a pair of different hash values. Because of this, such vectors will not be considered similar by the algorithm. This aspect results in a higher rate of *false negatives*.

**Theorem.** Consider  $r$  independent  $(s_1, s_2, p_1, p_2)$ -sensitive LSH functions. Combining these  $r$  LSH functions with the *AND* operator results in a new LSH



*function with parameters as*

$$H' = (d_1, d_2, \rho_1^r, \rho_2^r)$$

It is easy to prove this statement by using the probability formula of several independent events which multiplies the probabilities of all events to estimate the probability that all events will occur.

### OR operator

*Given  $n$  independent LSH functions  $H_1, H_2, \dots, H_n$ , the OR operator considers two vectors as a candidate pair only if **at least one** of  $n$  corresponding hash values of both vectors are equal. Otherwise, the vectors are not considered as candidates.*

Inversely to the AND operator, the OR operator increases the probability of any two vectors of being candidates. For any pair of vectors, it is sufficient at least a single equality of corresponding hash values. For that reason, the OR aggregation decreases the number of *false negatives* and increases *false positives*.

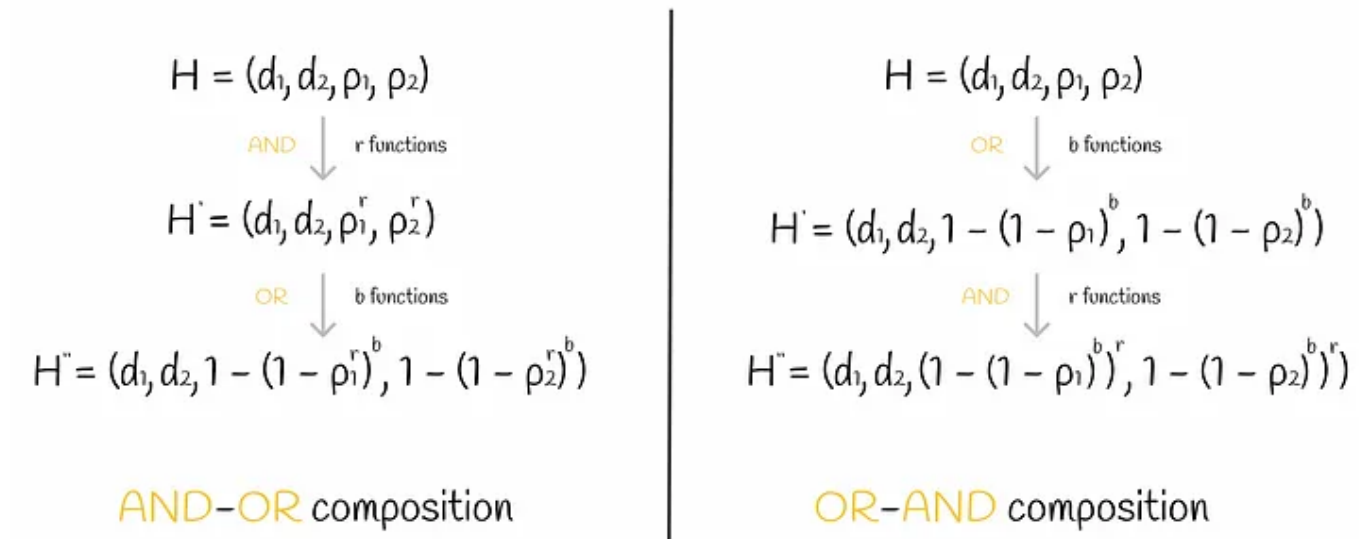
**Theorem.** *Consider  $b$  independent  $(d_1, d_2, p_1, p_2)$ -family LSH functions. Combining these  $b$  LSH functions with the AND operator results in a new LSH function with parameters as*

$$H' = (d_1, d_2, 1 - (1 - \rho_1)^b, 1 - (1 - \rho_2)^b)$$

We will not prove this theorem because the resulting analogous probability formula was obtained and explained in part 5 of this article series.

## Composition

Having *AND* and *OR* operations, it is possible to combine them together in various manners to better control the *false positive* and *false negative* rates. Let us imagine having  $r$  LSH functions used by the *AND* combinator and  $b$  LSH functions used by the *OR* combinator. Two different compositions can be constructed by using these basic combinators:



AND-OR and OR-AND are two types of compositions that can be built by using AND and OR operators.

Algorithms described in two previous articles used the *AND-OR* composition. In fact, nothing prevents us from building more complex compositions based on *AND* and *OR* operations.

## Composition example

Let us study an example to figure out how combinations of *AND* and *OR* can significantly improve performance. Assume an *OR-AND* combination with parameters  $b = 4$  and  $r = 8$ . Based on the corresponding formula above, we

can estimate how the initial probability of two vectors being candidates transforms after the composition:

$\rho, \%$	0	10	20	30	40	50	60	70	80	90	100
$\rho', \%$	0	0.02	1.48	11.1	32.9	59.7	81.3	72.4	98.7	99.9	100

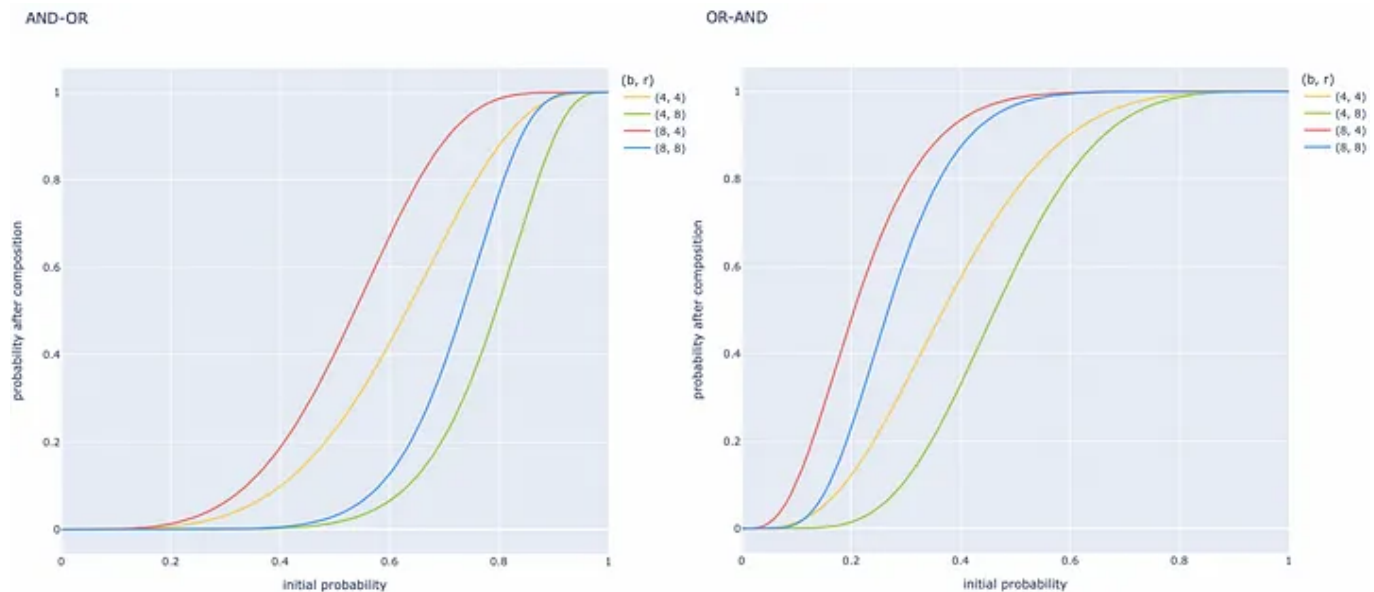
Probability changes by applying an OR-AND composition with parameters  $b = 4$  and  $r = 8$ . The first row shows initial probabilities while the second row shows the transformed ones.

For instance, if for a certain value of similarity between two vectors, a single LSH function hashes them to the same bucket in 40% of cases, then after the OR-AND composition they will be hashed in 32.9% of cases.

To understand what is so special about compositions, consider a  $(0.4, 1.7, 0.8, 0.2)$ -sensitive LSH function. After the OR-AND transformation, the LSH function transforms into  $(0.4, 1.7, 0.0148, 0.987)$ -sensitive format.

In essence, if initially two vectors were very similar and had a distance of less than 0.4, then they would be considered candidates in 80% of cases. However, with the composition applied, they are now candidates in 98.7% of scenarios resulting in much fewer *false negative* errors!

Analogously, if two vectors are very different from each other and have a distance greater than 1.7, then they are now considered candidates only in 1.48% of cases (compared to 20% before). This way, the frequency of *false positive* errors is reduced by 13.5 times! This is a massive improvement!



Curves showing how initial probabilities are transformed after different compositions

*In general, by having a  $(d_1, d_2, p_1, p_2)$ -sensitive LSH function, it is possible to transform it to a  $(d_1, d_2, p'_1, p'_2)$  format where  $p'_1$  is close to 1 and  $p'_2$  is close to 0. Getting more closer to 1 and 0 usually requires more compositions to be used.*

## LSH for other distance metrics

We have already studied in-depth LSH schemes which are used to preserve information about Jaccard index and cosine distance. The natural question which arises is whether it is possible to use LSH for other distance metrics. Unfortunately, for most of the metrics, there is no corresponding LSH algorithm.

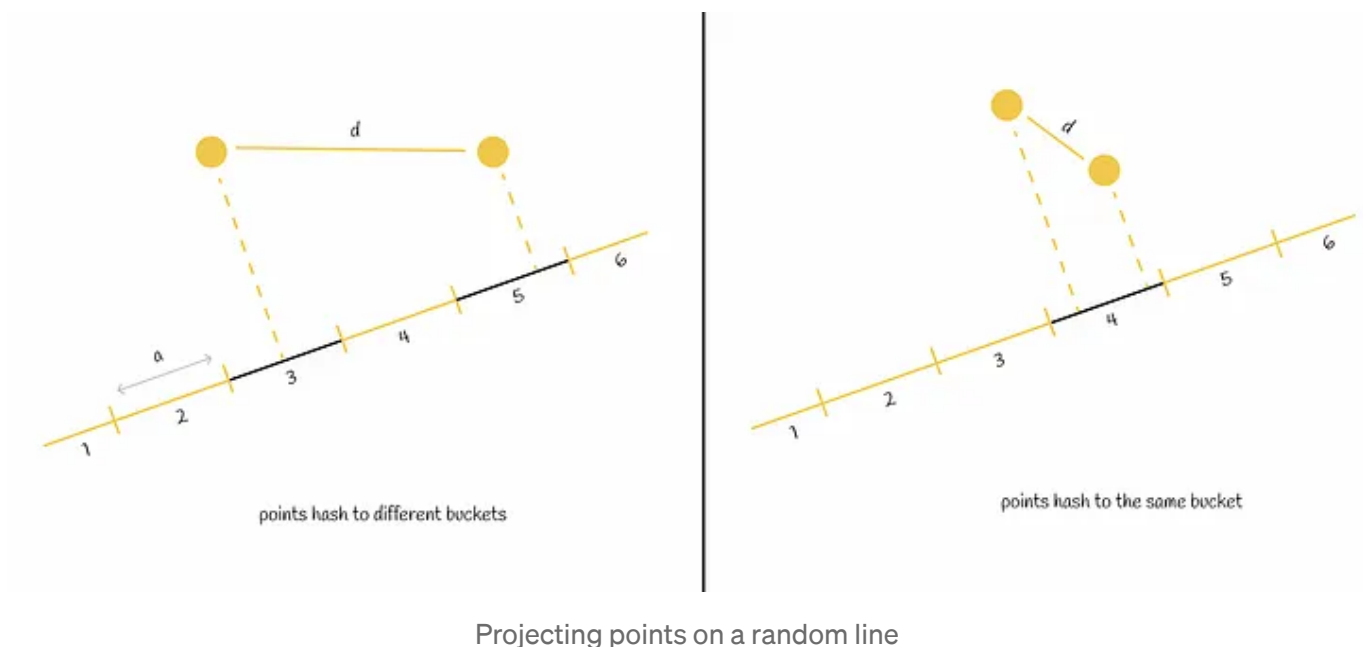
Nevertheless, LSH schema exists for Euclidean distance — one of the most commonly used metrics in machine learning. As it is used frequently, we are going to study how to get hash values for Euclidean distance. With the theoretical notations introduced above, we will prove an important LSH property for this metric.

## LSH for Euclidean distance

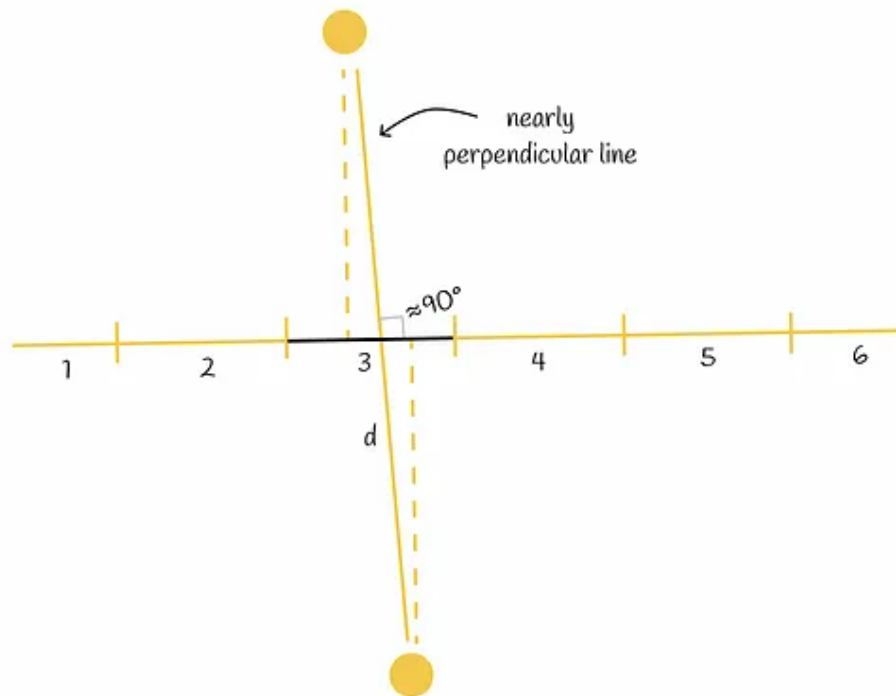
The mechanism of hashing of points in Euclidean space includes projecting them on a random line. Algorithm assumes that

- If two points are relatively close to each other, then their projections should also be close.
- If two points are far from each other, then their projections should be also far.

To measure how close two projections are, a line can be divided into several equal segments (buckets) of size  $a$  each. Each line segment corresponds to a certain hash value. If two points project to the same line segment, then they have the same hash value. Otherwise, hash values are different.



Though the method might seem robust at first, it can still project points which are far from each other to the same segment. This happens especially when a line connecting two points is almost perpendicular to the initial projection line.



Despite both points being relatively far from each other, there is still chance that they will be hashed into the same bucket.

In order to decrease the error rate, it is highly recommended to use compositions of random projection lines, as discussed above.

It is geometrically possible to prove that if  $a$  is a length of a single line segment in Euclidean space, then  $H$  is  $(a/2, 2a, 1/2, 1/3)$ -sensitive LSH function.

## Conclusion

In this chapter, we accumulated knowledge on general LSH notation which helped us to formally introduce composition operations allowing us significantly reduce error rates. It is worth noting that LSH exists only for a small part of machine learning metrics but at least for the most popular ones which are Euclidean distance, cosine distance and Jaccard index. When dealing with another metric measuring similarity between vectors, it is recommended to choose another similarity search approach.

For reference, formal proofs of the statements introduced in this article can be found in [these notes](#).

## Resources

- [Locality Sensitive Hashing](#) | [Lecture Notes for Big Data Analytics](#) | [Nimrah Mustafa](#)
- [Cosine distance](#) | [Wikipedia](#)

*All images unless otherwise noted are by the author.*

[Machine Learning](#)[Similarity Search](#)[Data Science](#)[Lsh](#)[Metrics](#)

---

### More from the list: "NLP"

Curated by Himanshu Birla



Jon Gi... in Towards Data ...

#### Characteristics of Word Embeddings

★ · 11 min read · Sep 4, 2021



Jon Gi... in Towards Data ...

#### The Word2vec Hyperparameters

★ · 6 min read · Sep 3, 2021



Jon Gi... in

#### The Word2vec

★ · 15 min read



[View list](#)





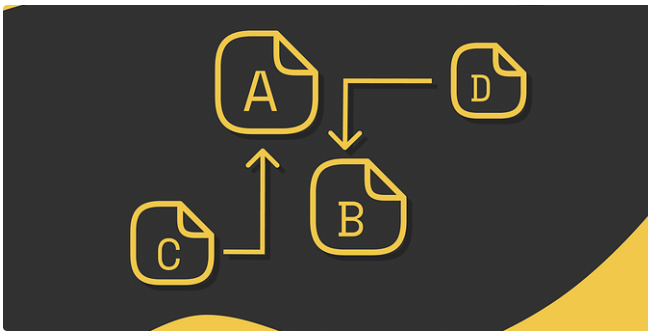
## Written by Vyacheslav Efimov

Following

470 Followers · Writer for Towards Data Science

BSc in Software Engineering. Passionate machine learning engineer. Writer at Towards Data Science.

### More from Vyacheslav Efimov and Towards Data Science



 Vyacheslav Efimov in Towards Data Science

### Visualised Explanation of PageRank

Discover how Google search engine ranks documents based on their link structure

14 min read · Aug 10




40



1



 Antonis Makropoulos in Towards Data Science

### How to Build a Multi-GPU System for Deep Learning in 2023

This story provides a guide on how to build a multi-GPU system for deep learning and...

10 min read · Sep 17



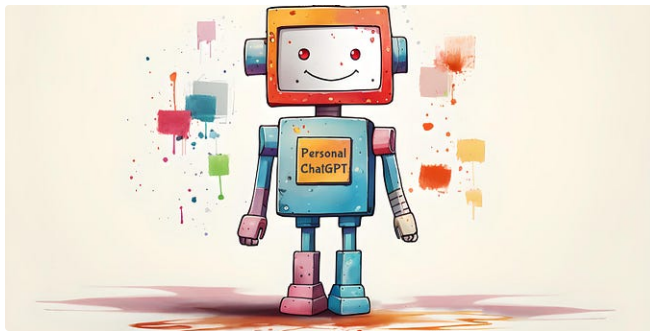
549



11







 Robert A. Gonsalves in Towards Data Science

## Your Own Personal ChatGPT

How you can fine-tune OpenAI's GPT-3.5 Turbo model to perform new tasks using you...

🌟 · 15 min read · Sep 8



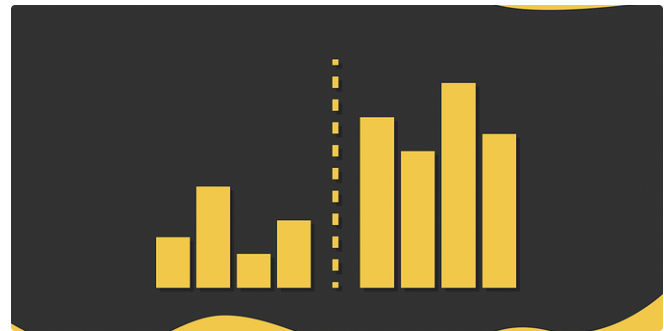
595



7



13



 Vyacheslav Efimov in Towards Data Science

## Overview of Sorting Algorithms: Quicksort

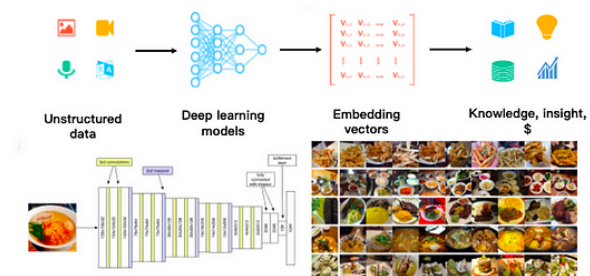
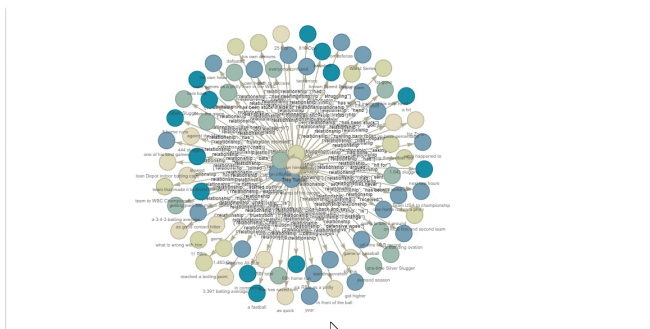
Quicksort is probably the most popular sorting algorithm. In many experiments, it w...


5 min read · Dec 30, 2022

See all from Vyacheslav Efimov

See all from Towards Data Science

## Recommended from Medium



 Wenqi Glantz in Better Programming

## 7 Query Strategies for Navigating Knowledge Graphs With...

Exploring NebulaGraph RAG Pipeline with the Philadelphia Phillies

★ · 17 min read · 4 days ago

 501  4

 ...

 Jayita Bhattacharyya in GoPenAI

## Primer on Vector Databases and Retrieval-Augmented Generation...

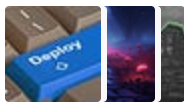
Vector Databases Generation (RAG)  
Langchain Pinecone HuggingFace Large...

9 min read · Aug 16

 228  1

 ...

### Lists



#### Predictive Modeling w/ Python

20 stories · 452 saves



#### Practical Guides to Machine Learning

10 stories · 519 saves



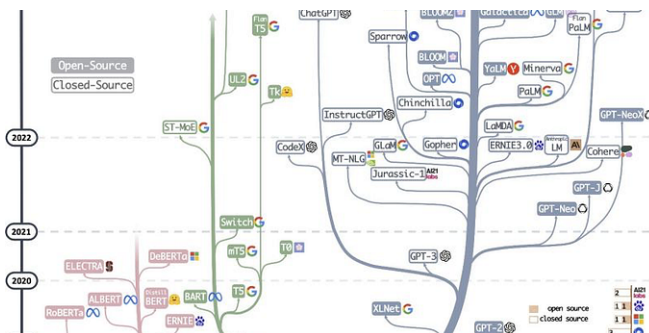
#### Natural Language Processing

669 stories · 283 saves



#### New Reading List

174 stories · 133 saves



 Haifeng Li

## A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's...

15 min read · Sep 14

 Han HELOIR, Ph.D. in Artificial Corner

## MongoDB and Langchain Magic: Your Beginner's Guide to Setting...

Introduction:

★ · 7 min read · Sep 12



372



1.4K



12



Ray Serve



Amazon SageMak



Behnaz Nojavanasghari

## Top 10 Tools for Deploying Machine Learning Models to Production:...

Introduction

11 min read · Jun 22



35



71



1



Alyx

## Semantic Search with FAISS

HuggingFace get\_nearest\_example and Cosine Similarity Search

9 min read · Jul 15

[See more recommendations](#)