



Search Medium

Write



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

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



Vyacheslav Efimov · Following

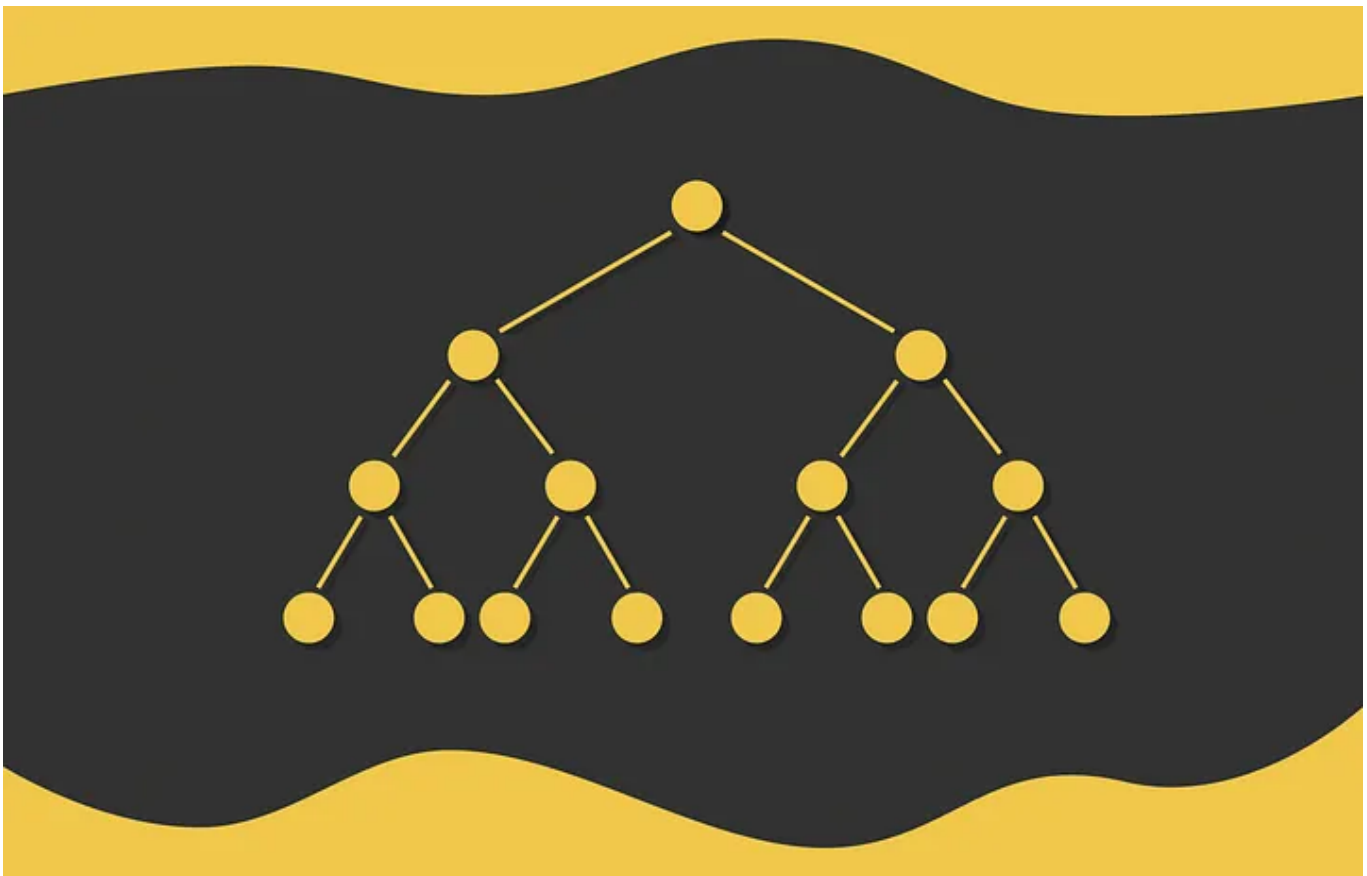
Published in Towards Data Science · 12 min read · Jul 21



87



1



**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 part, we looked at the main paradigm of LSH which is to *transform input vectors to lower-dimensional hash values while preserving information about their similarity*. For obtaining hash values (signatures) minhash functions were used. In this article, we are going to randomly project input data to get analogous binary vectors.

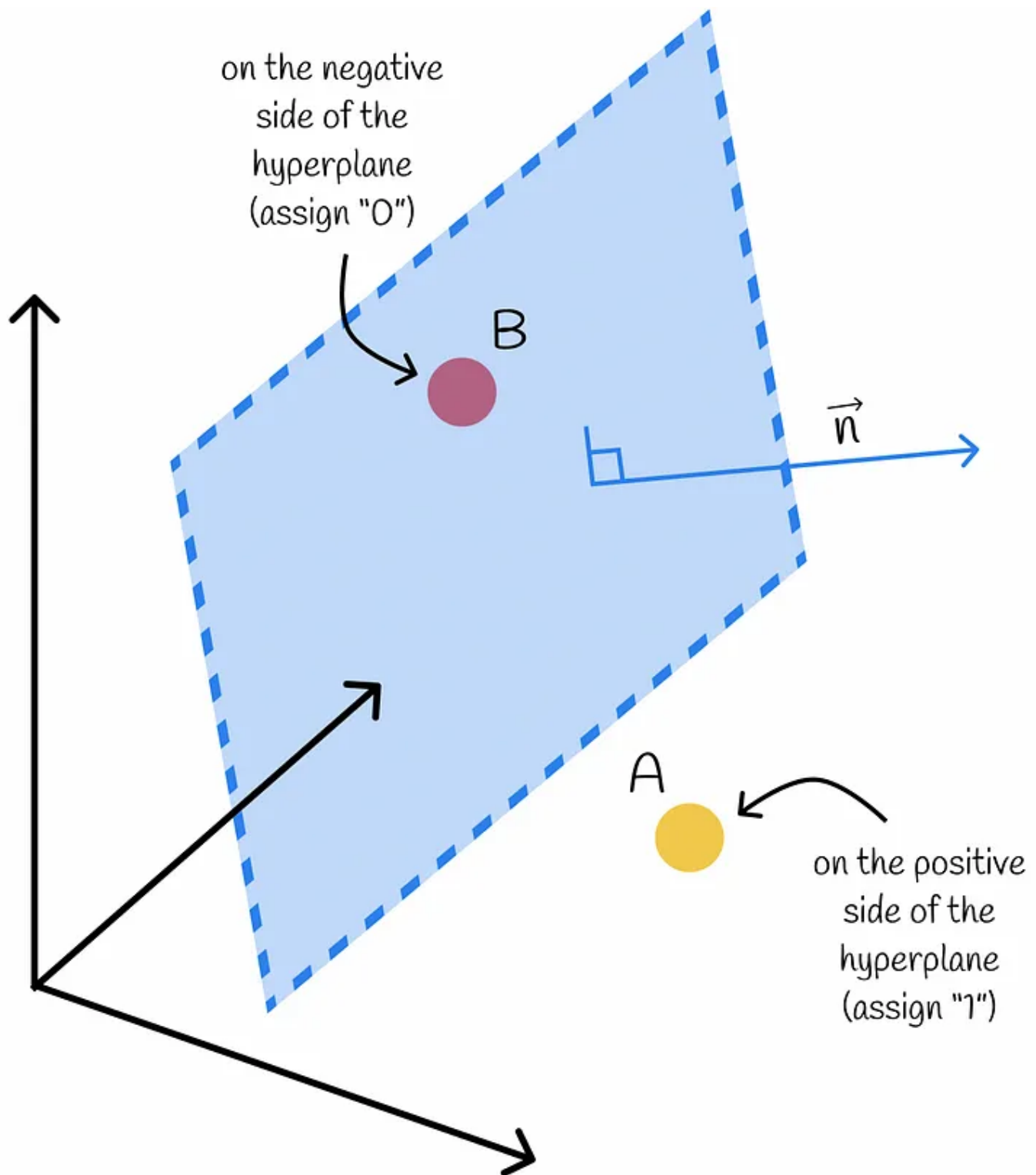
### Similarity Search, Part 5: Locality Sensitive Hashing (LSH)

Explore how similarity information can be incorporated into hash function

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

## Idea

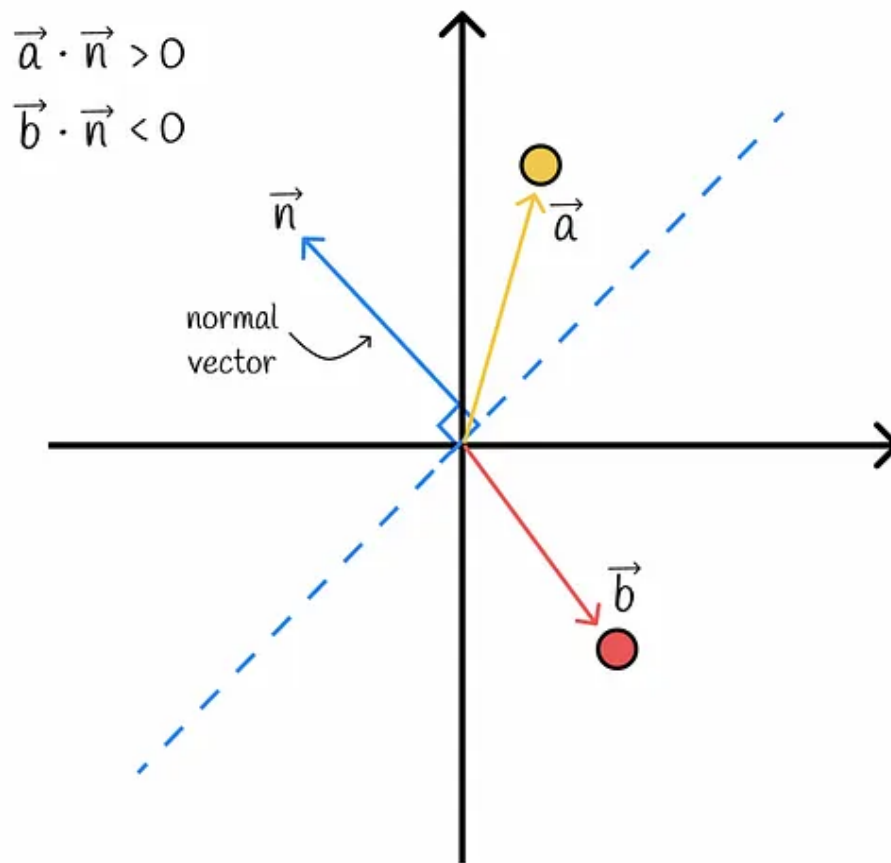
Consider a group of points in a high-dimensional space. It is possible to construct a random hyperplane that acts as a wall and separates each point into one of two subgroups: positive and negative. A value of “1” is assigned to each point on the positive side and “0” to each point on the negative side.



Example of a hyperplane separating two points in 3D space

How can the side of a hyperplane for a certain vector be determined? By using the inner product! Getting to the essence of linear algebra, the sign of the dot product between a given vector and the normal vector to the

hyperplane determines on which side the vector is located. This way, every dataset vector can be separated into one of two sides.



Calculating the inner product of a vector with the normal vector of a hyperplane and comparing it with 0 can tell on which side the vector is located relative to the hyperplane

Obviously, encoding every dataset vector with one binary value is not sufficient. That is several random hyperplanes should be constructed, so every vector can be encoded with that many values of 0 and 1 based on its relative position to a specific hyperplane. If two vectors have absolutely the same binary code, it indicates that none of the constructed hyperplanes could have separated them into different regions. Thus, they are likely to be very close to each other in reality.

For finding the nearest neighbour for a given query, it is sufficient to encode the query with 0s and 1s in the same way by checking its relative positions to

all the hyperplanes. The found binary vector for the query can be compared to all other binary vectors obtained for dataset vectors. This can be done linearly by using the Hamming distance.

*Hamming distance between two vectors is the number of positions at which their values are different.*

1	0
0	0
0	1
1	1
0	0

Hamming distance = 2

0	1
1	0
1	1
0	1
0	1

Hamming distance = 4

Example of computing Hamming distance. A pair of vectors on the left are more similar to each other since their Hamming distance is smaller.

The binary vectors with the least Hamming distances to the query are taken as candidates and then fully compared to the initial query.

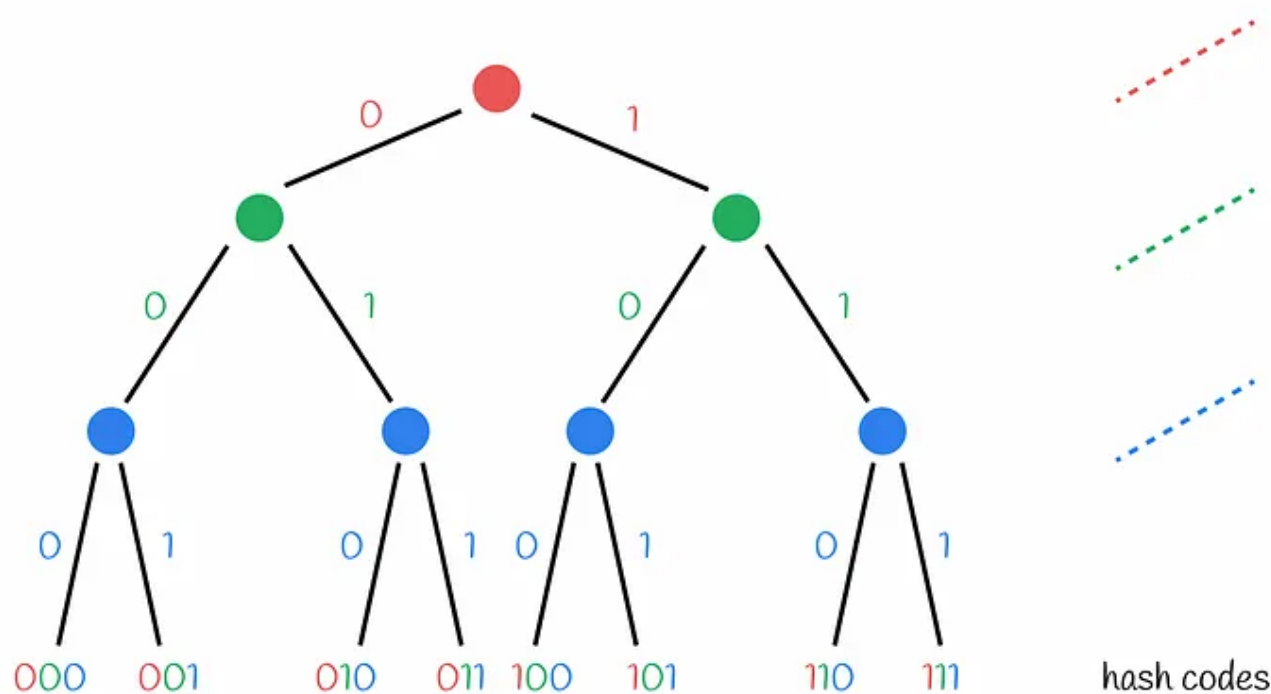
### Why are hyperplanes built randomly?

At the current stage, it seems logical to demand why hyperplanes are built in a random manner and not in deterministic meaning that custom rules for separating dataset points could have been defined. There are two principal reasons behind this:

- Firstly, the deterministic approach is not able to generalize the algorithm and can lead to overfitting.
- Secondly, randomness allows for making probabilistic statements regarding the algorithm's performance which is not dependent on the input data. For a deterministic approach, this would not work out because it might act well on one data and have a poor performance on another. A good analogy for this is the deterministic quick-sort algorithm which works on average in  $O(n * \log n)$  of time. However, it works for  $O(n^2)$  of time on a sorted array as the worst-case scenario. If somebody has knowledge about the algorithm's workflow, then this information can be used to expressively damage the efficiency of the system by always providing the worst-case data. That is why a randomized quick-sort is much more preferred. The absolutely similar situation occurs with random hyperplanes.

### **Why are LSH random projections also called “trees”?**

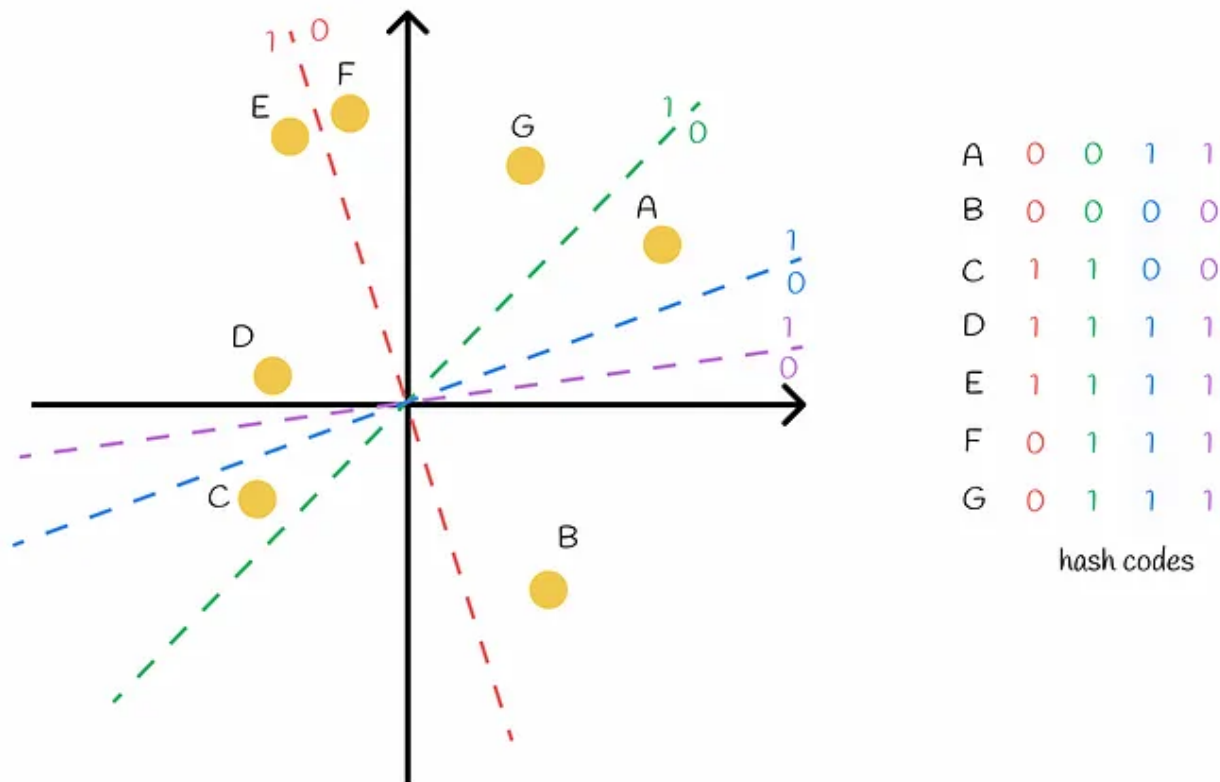
The random projections method is sometimes called **LSH Tree**. This is due to the fact that the process of hash code assignment can be represented in the form of the decision tree where each node contains a condition of whether a vector is located on the negative or positive side of a current hyperplane.



The first node checks on which side a vector is located relative to the red hyperplane. Nodes on the second level check the same condition but relative to the green hyperplane. Finally, the third level checks the relative position of the blue hyperplane. Based on these 3 conditions, the vector is assigned a 3-bit hash.

## Forest of hyperplanes

Hyperplanes are constructed randomly. This may result in a scenario when they poorly separate dataset points which is shown in the figure below.



4 hyperplanes are constructed to represent dataset points as 4-length binary vectors. Even though points D and E have the same hash code, they are relatively far from each other (FP). The inverse situation occurs with a pair of points E and F which are located in different regions but are very close to each other (FN). Taking into consideration the Hamming distance, the algorithm normally predicts point D as being closer to E than point F.

Technically, it is not a big deal when two points have the same hash code but are far from each other. In the next step of the algorithm, these points are taken as candidates and are fully compared — this way the algorithm can eliminate *false positive* cases. The situation is more complicated with *false negatives*: when two points have different hash codes but in reality are close to each other.

Why not use the same approach with decision trees from classical machine learning which are combined into random forests to improve the overall prediction quality? *If one estimator commits an error, other estimators can produce better predictions and alleviate the final prediction error.* Using this idea, the process of building random hyperplanes can be independently repeated.



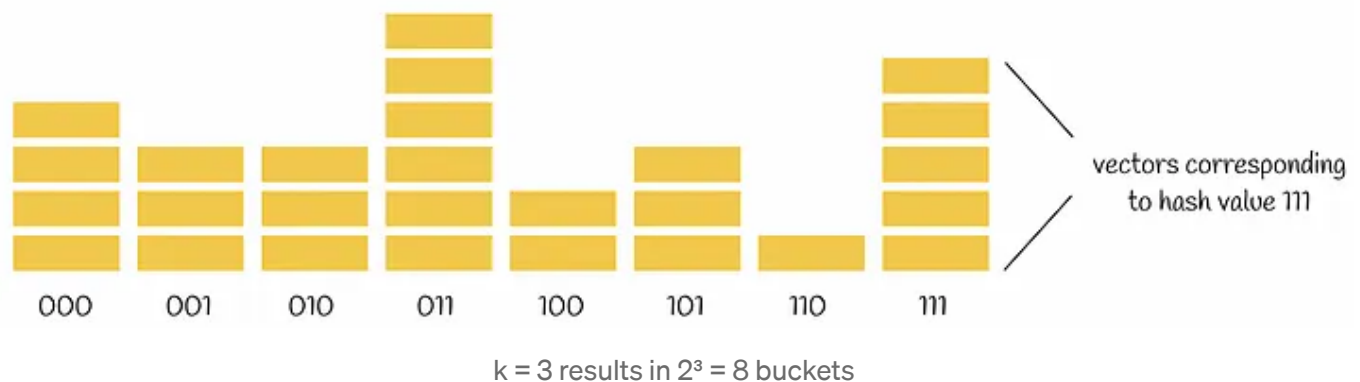
Resulting hash values can be aggregated for a pair of vectors in a similar way to how it was with minhash values in the previous chapter:

If a query has the same hash code at least once with another vector, then they are considered candidates.

Using this mechanism the number of *false negatives* can be decreased.

## Quality vs speed trade-off

It is important to choose an appropriate number of hyperplanes to run on the dataset. The more hyperplanes are chosen to partition dataset points, the fewer collisions there are and the more time it takes to compute hash codes and more memory to store them. Speaking exactly, if a dataset consists of  $n$  vectors and we split it by  $k$  hyperplanes, then on average every possible hash code will be assigned to  $n / 2^k$  vectors.



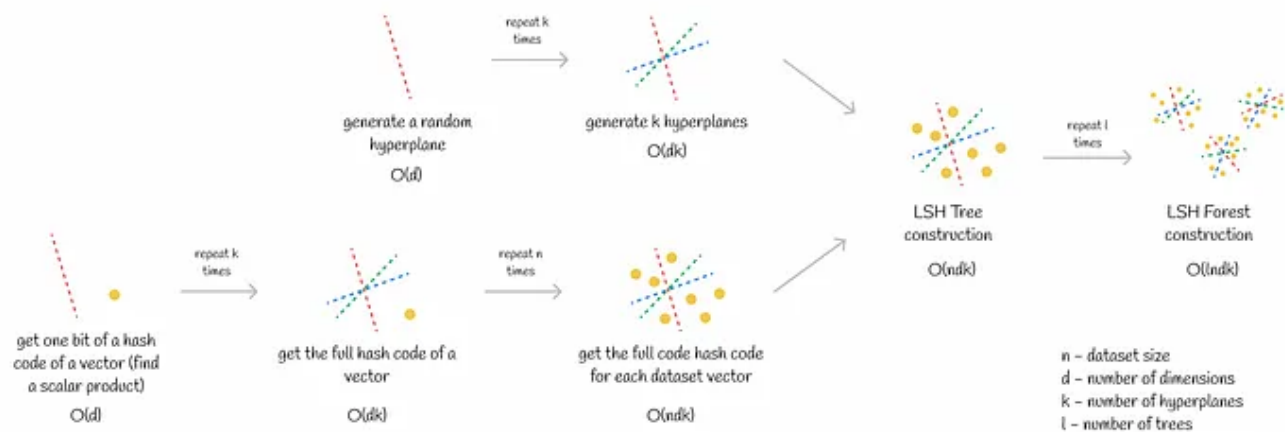
## Complexity

### Training

The LSH Forest training phase consists of two parts:

1. *Generation of  $k$  hyperplanes.* This is a relatively fast procedure since a single hyperplane in  $d$ -dimensional space can be generated in  $O(d)$  of time.
2. *Assigning hash codes to all dataset vectors.* This step might take time, especially for large datasets. Obtaining a single hash code requires  $O(dk)$  of time. If a dataset consists of  $n$  vectors, then the total complexity becomes  $O(ndk)$ .

The process above is repeated several times for each tree in the forest.



Training complexity

## Inference

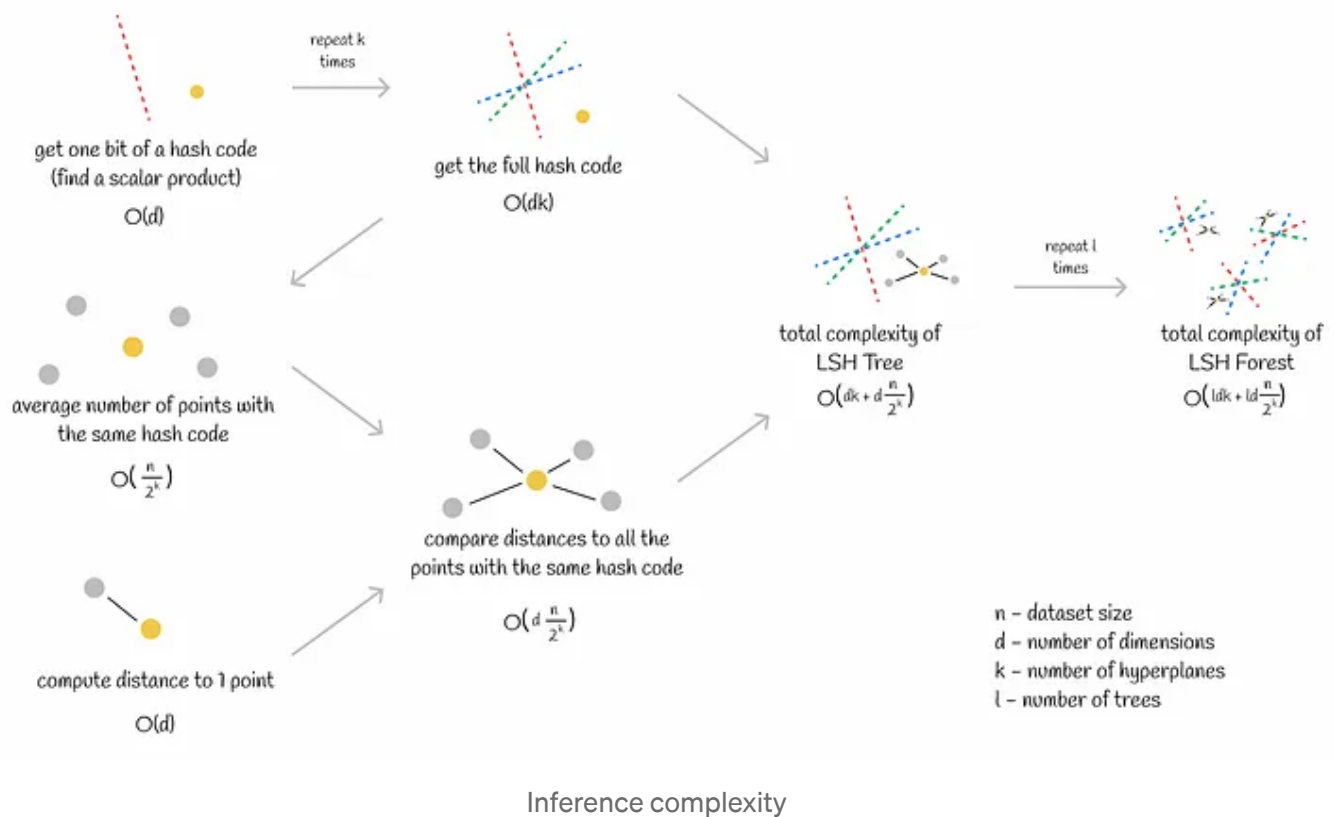
One of the advantages of LSH forest is its fast inference which includes two steps:

1. *Getting the hash code of a query.* This is equivalent to computing  $k$  scalar products which result in  $O(dk)$  of time ( $d$  — dimensionality).

2. *Finding the nearest neighbours* to the query within the same bucket (vectors with the same hash code) by calculating precise distances to the candidates. Distance computation proceeds linearly for  $O(d)$ . Every bucket on average contains  $n / 2^k$  vectors. Therefore, distance calculation to all the potential candidates requires  $O(dn / 2^k)$  of time.

The total complexity is  $O(dk + dn / 2^k)$ .

As usual, the process above is repeated several times for each tree in the forest.



When the number of hyperplanes  $k$  is chosen in such a way that  $n \sim 2^k$  (which is possible in most cases), then the total inference complexity becomes  $O(ldk)$  ( $l$  is the number of trees). Basically, this means that **the computational time does not depend on the dataset size!** This subtlety

results in efficient scalability of similarity search for millions or even billions of vectors.

## Error rate

In the previous part of the article about LSH, we discussed how to find the probability that two vectors will be chosen as candidates based on their signature similarity. Here we are going to use almost the same logic to find the formula for LSH forest.

$$p = s$$

Let  $s$  be the probability that two vectors have the same bit at the same position of their hash values ( $s$  will be estimated later)

$$p = s^k$$

The probability that hash codes of length  $k$  of two vectors are equal

$$p = 1 - s^k$$

The probability that hash codes of length  $k$  of two vectors are different (or at least one bit is different)

$$p = (1 - s^k)^l$$

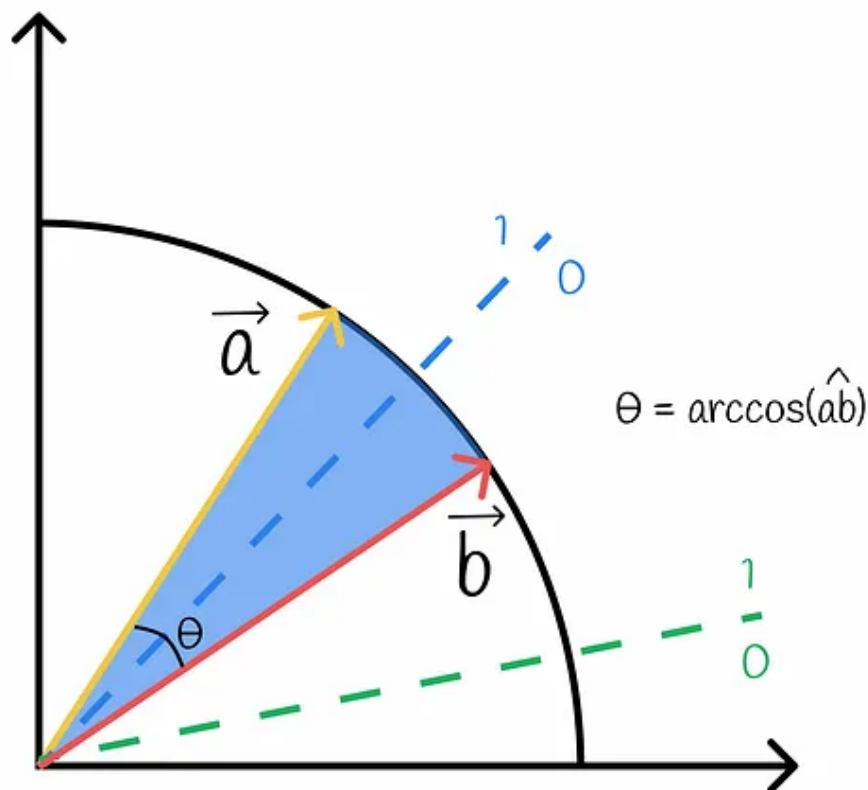
The probability that all  $l$  hash codes (for  $l$  hyperplanes) of two vectors are different

$$P = 1 - (1 - s^k)^l$$

The probability that at least one of  $l$  hash codes of two vectors is equal, so the vectors will become candidates

So far we have almost obtained the formula for estimating the probability of two vectors being candidates. The only thing left is to estimate the value of the variable  $s$  in the equation. In the classic LSH algorithm,  $s$  is equal to the Jaccard index or signature similarity of two vectors. On the other hand, for estimating  $s$  for LSH forest, linear algebra theory is going to be used.

Frankly speaking,  $s$  is the probability that two vectors  $a$  and  $b$  have the same bit. This probability is equivalent to the probability that a random hyperplane separates these vectors to the same side. Let us visualise it:



Vectors  $a$  and  $b$  are separated by the blue hyperplane. The green hyperplane does not separate them.

From the figure, it is clear that a hyperplane separates vectors  $a$  and  $b$  into two different sides only in case when it passes between them. Such probability  $q$  is proportional to the angle between the vectors which can be easily computed:

$$q = \frac{\arccos(ab)}{\pi/2}$$

The probability that a random hyperplane separates two vectors (i.e. so they have different bits)

$$s = 1 - 2 \frac{\arccos(ab)}{\pi}$$

The probability that a random hyperplane does not separate two vectors (i.e. so, they have the same bit)

Plugging this equation into the one which was obtained previously leads to the final formula:

$$p = 1 - \left( 1 - \left( 1 - 2 \frac{\arccos(ab)}{\pi} \right)^k \right)^l$$

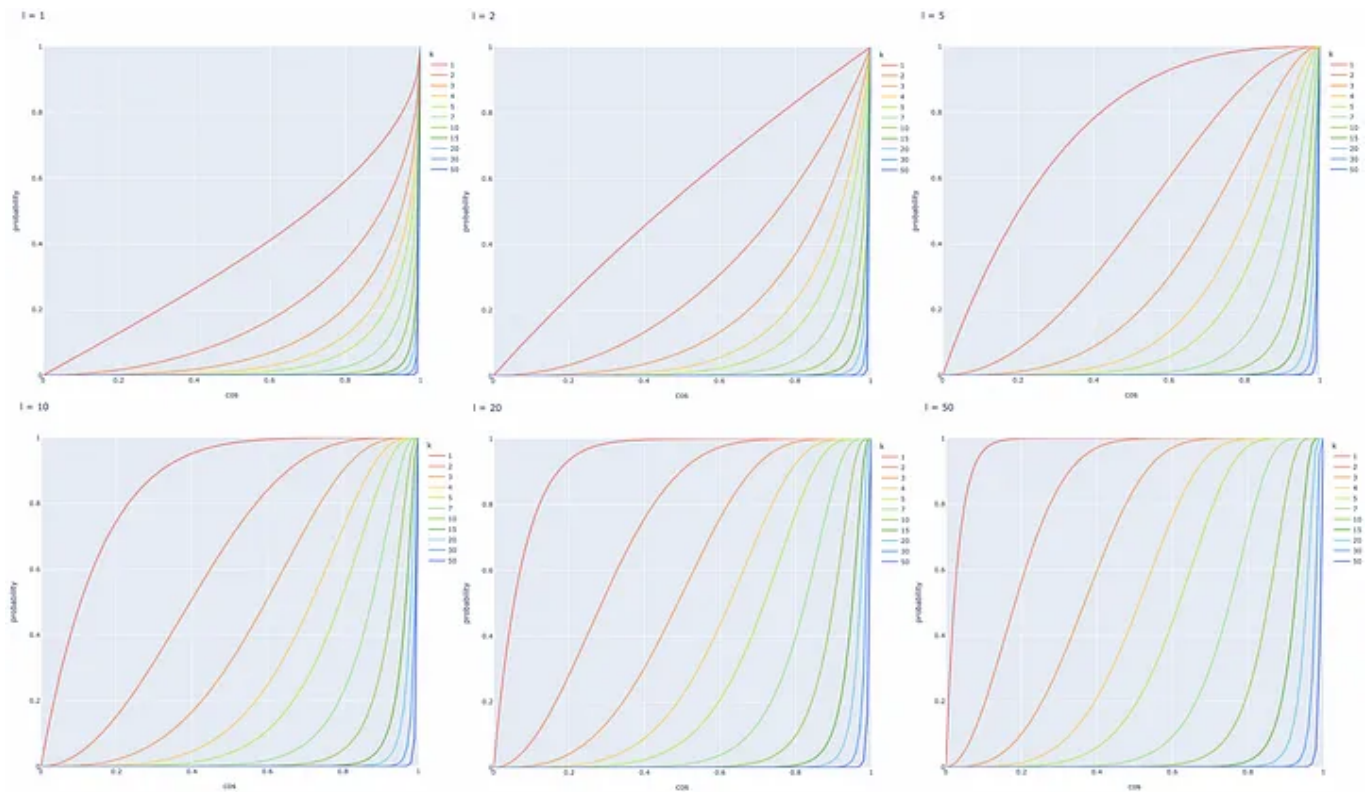
The probability that two vectors have at least one corresponding hash value (i.e. so they become candidates) based on the number of hyperplanes  $k$  and the number of LSH trees  $l$

## Visualisation

*Note.* Cosine similarity is formally defined in range  $[-1, 1]$ . For simplicity, we will map this interval to  $[0, 1]$  where 0 and 1 indicate the lowest and the

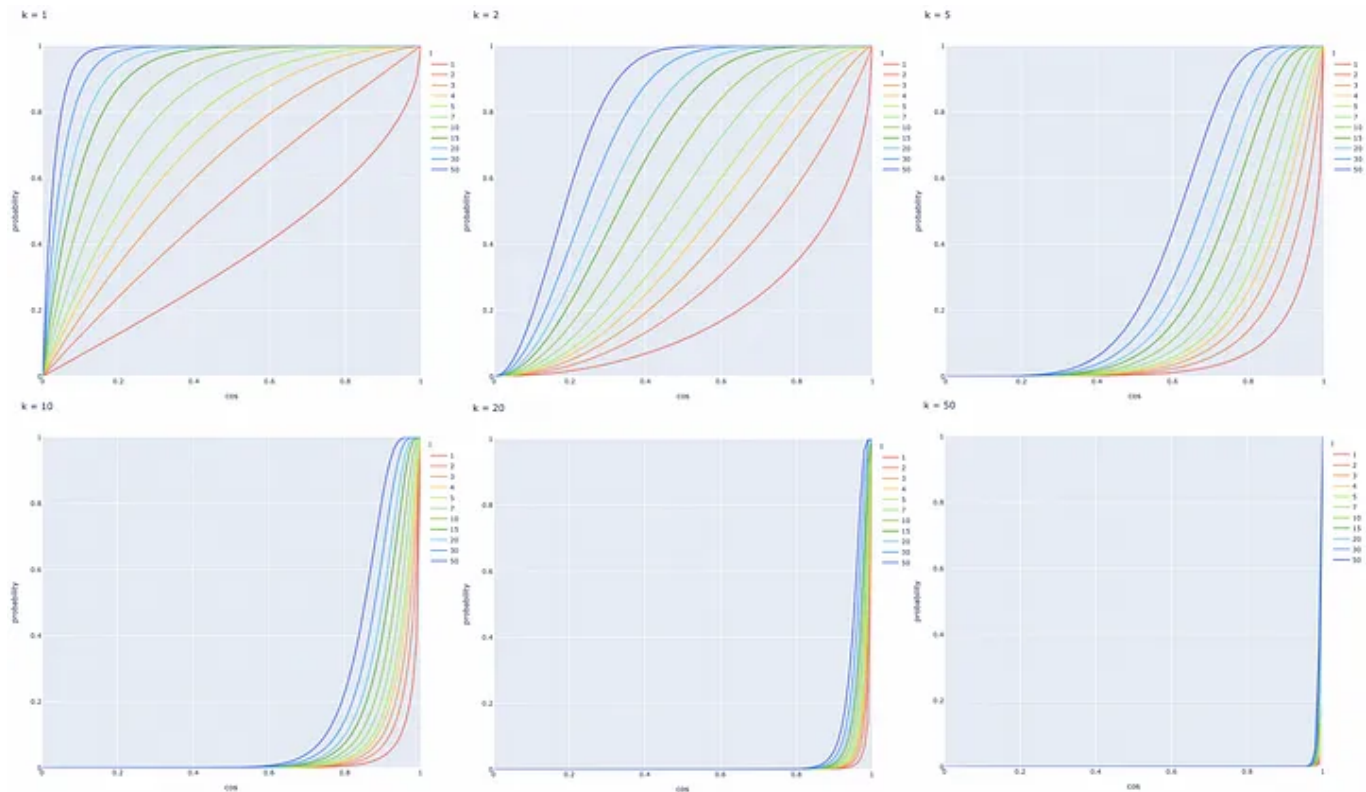
highest possible similarity respectively.

With the last obtained formula, let us visualise the probability of two vectors being candidates based on their cosine similarity for a different number of hyperplanes  $k$  and trees  $l$ .



Adjusting number of trees  $l$



Adjusting number of hyperplanes  $k$ 

Several useful observations can be made, based on the plots:

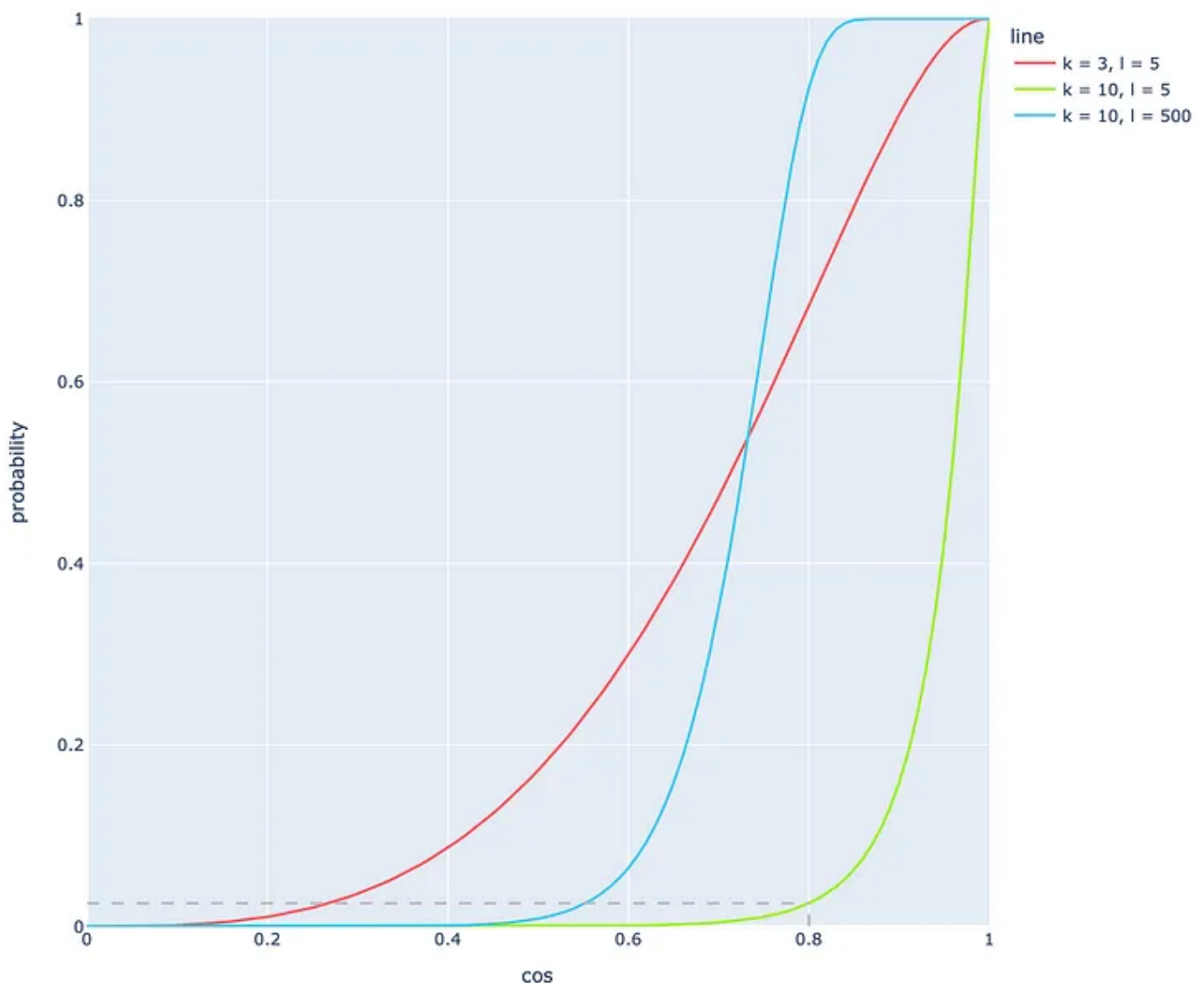
- A pair of vectors with the cosine similarity of 1 always become candidates.
- A pair of vectors with the cosine similarity of 0 never become candidates.
- The probability  $P$  of two vectors being candidates increases (i.e. more *false positives*) when the number of hyperplanes  $k$  decreases or the number of LSH trees  $l$  increases. The inverse statement is true.

To summarise things up, LSH is a very flexible algorithm: it is possible to adjust different values  $k$  and  $l$  based on a given problem and acquire the probability curve that satisfies the problem's requirements.

## Example



Let us look at the following example. Imagine  $l = 5$  trees are constructed with  $k = 10$  hyperplanes. Apart from it, there are two vectors with the cosine similarity of 0.8. In most systems, such cosine similarity indicates that the vectors are indeed very close to each other. Based on the results in previous sections, it turns out that this probability equals only 2.5%! Obviously, it is a very low result for such a high cosine similarity. Using these parameters of  $l = 5$  and  $k = 10$  results in a huge number of *false negatives*! The green line below represents the probability in this case.



Probability curve based on the cosine similarity of two vectors

This issue can be solved by adjusting better values of  $k$  and  $l$  to move the curve to the left.

For instance, if  $k$  is decreased to 3 (red line), then the same value of the cosine similarity of 0.8 will correspond to the probability of 68% which is better than before. At first sight, it seems like the red line fits much better than the green one but it is important to keep in mind that using small values of  $k$  (as in the case with the red line) leads to an enormous number of collisions. That is why it is sometimes preferable to adjust the second parameter which is the number of trees  $l$ .

Unlike with  $k$ , it usually requires a very high number of trees  $l$  to get a similar line shape. In the figure, the blue line is obtained from the green one by changing the value of  $l$  from 10 to 500. The blue line clearly fits better than the green one but it is still far from being perfect: because of a high slope between cosine similarity values of 0.6 and 0.8, the probability is almost equal to 0 around cosine similarity of 0.3-0.5 which is unfavorable. This small probability for a document similarity of 0.3–0.5 should normally be higher in real life.

Based on the last example, it is evident that even a very high number of trees (which require lots of computations) still results in many *false negatives*! That is the main disadvantage of the random projections method:

*Though it is potentially possible to get a perfect probability curve, it would either require lots of computations or would result in lots of collisions. Otherwise, it leads to a high false negative rate.*

## Faiss implementation

***Faiss** (Facebook AI Search Similarity) is a Python library written in C++ used for optimised similarity search. This library presents different types of indexes which are data structures used to efficiently store the data and perform queries.*

Based on the information from the [Faiss documentation](#), we will find out how to build LSH index.

The random projections algorithm is implemented in Faiss inside the *IndexLSH* class. Though the Faiss authors use a slightly another technique called “random rotations”, it still has similarities with what was described in this article. The class implements only a single LSH tree. If we want to use an LSH forest, then it is enough just to create several LSH trees and aggregate their results.

The constructor of the *IndexLSH* class takes two arguments:

- **d**: the number of dimensions
- **nbits**: the number of bits required to encode a single vector (the number of possible buckets equals  $2^{nbits}$ )

*Distances returned by the `search()` method are Hamming distances to the query vector.*

```
1 d = 256 # data dimension
2 nbits = 16 # length of hash codes
3
4 index = faiss.IndexLSH(d, nbits)
5
6 index.add(data)
7 buckets = faiss.vector_to_array(index.codes) # get hash values for each dataset vector
8
9 k = 3 # how many nearest neighbours to search for
10 D, I = index.search(query, k)
```

Faiss implementation of IndexLSH

Additionally, Faiss allows inspecting encoded hash values for each dataset vector by calling the `faiss.vector_to_array(index.codes)` method.

Since every dataset vector is encoded by *nbits* binary values, the number of bytes required to store a single vector is equal to:

$$\text{bytes} = \left\lceil \frac{\text{nbits}}{8} \right\rceil$$

## Johnson-Lindenstrauss lemma

Johnson-Lindenstrauss lemma is a fabulous lemma related to dimensionality reduction. While it may be difficult to fully understand its original statement, it can be formulated in simple words:

Choosing a random subset and projecting original data on it preserves corresponding pairwise distances between points.

To be more precise, having a dataset of  $n$  points, it is possible to represent them in a new space of  $O(\log n)$  dimensions in such a way that the relative distances between points will almost be preserved. If a vector is encoded by  $\sim \log n$  binary values in the LSH method, then the lemma can be applied. Moreover, LSH creates hyperplanes in a random manner exactly as the lemma requires.

What is also incredible about Johnson-Lindenstrauss lemma is the fact that **the number of dimensions of a new dataset does not depend on the dimensionality of the original dataset!** In practice, this lemma does not work well for very small dimensions.

## Conclusion

We have gone through a powerful algorithm for similarity search. Being based on a simple idea of separating points by random hyperplanes it usually performs and scales very well on large datasets. Moreover, it comes with good flexibility by allowing one to choose an appropriate number of hyperplanes and trees.

Theoretical results from Johnson-Lindenstrauss lemma reinforce the usage of the random projections approach.

## Resources

- [LSH Forest: Self-Tuning Indexes for Similarity Search](#)
- [The Johnson-Lindenstrauss Lemma](#)
- [Faiss documentation](#)
- [Faiss repository](#)
- [Summary of Faiss indexes](#)

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

Machine Learning

Similarity Search

Random Projection

Faiss

Thoughts And Theory

## 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

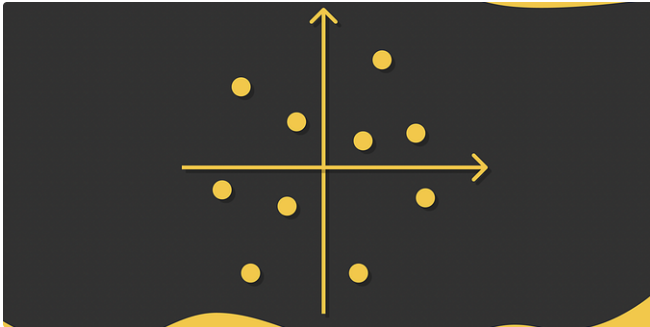
470 Followers · Writer for Towards Data Science

Following



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

### Similarity Search, Part 3: Blending Inverted File Index and Product...


In the first two parts of this series we have discussed two fundamental algorithms in...

8 min read · May 19



128



 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...

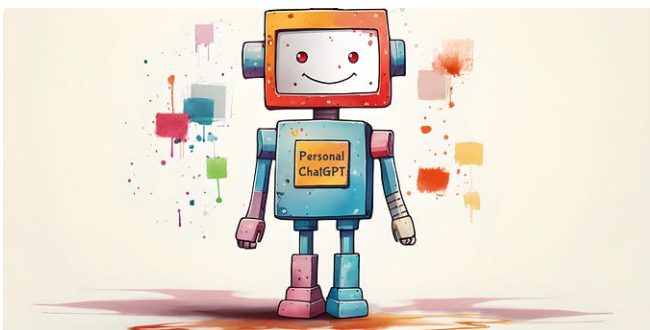
10 min read · Sep 17



549



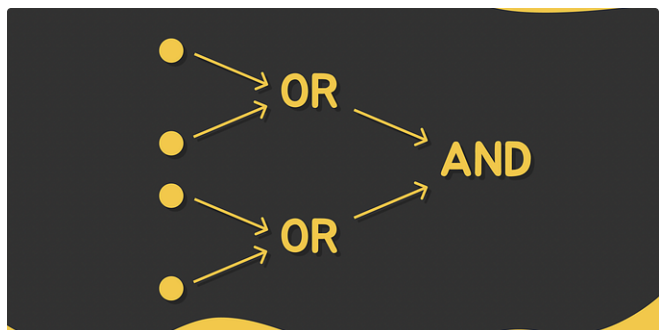
11




 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...



 Vyacheslav Efimov in Towards Data Science

### Similarity Search, Part 7: LSH Compositions

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

★ · 15 min read · Sep 8

11 min read · Jul 24



595



7



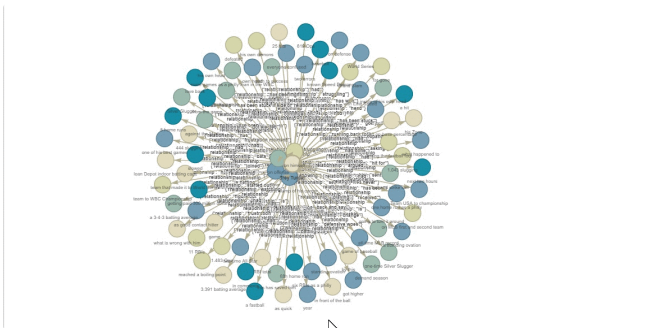
43



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



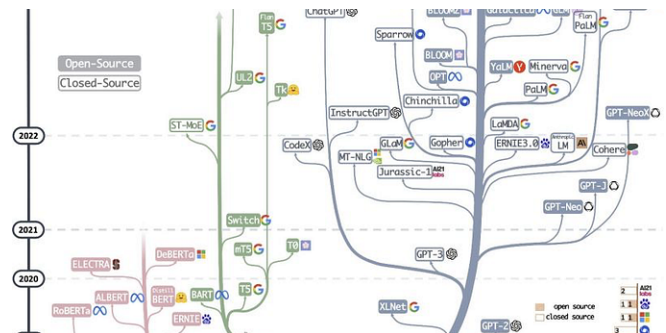
501



4



372



Haifeng Li

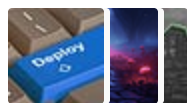
### A Tutorial on LLM

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

15 min read · Sep 14



## Lists



### Predictive Modeling w/ Python

20 stories · 452 saves



### Practical Guides to Machine Learning

10 stories · 519 saves



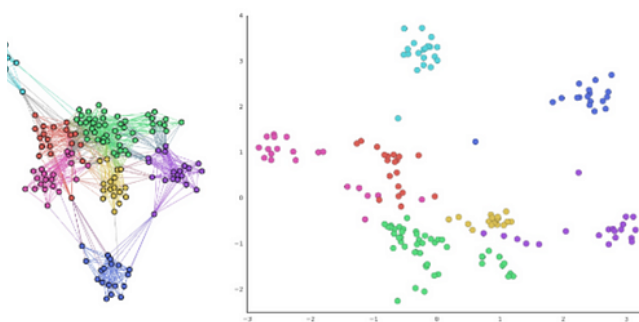
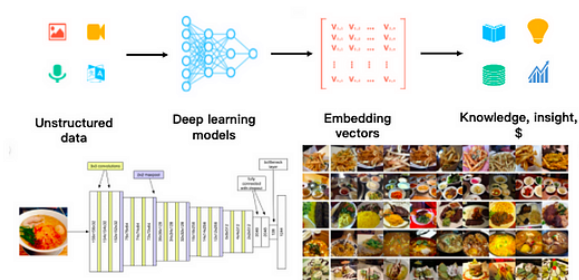
### Natural Language Processing

669 stories · 283 saves



### The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves



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



Artem Shlezinger

## Graph Embedding: Nonlinear Optimization

Simplified approach for finding vertex vectors

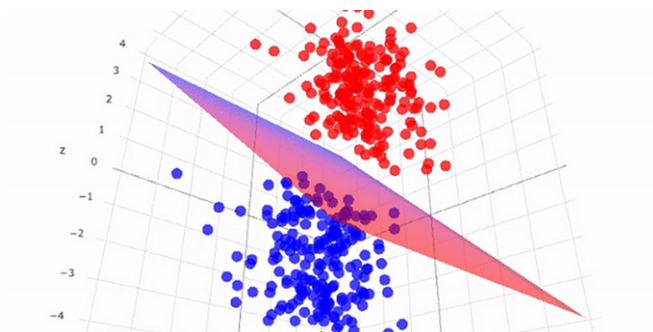
4 min read · Jul 30



9



1



Tasmay Pankaj Tibre... in Low Code for Data Scie...



Behnaz Nojavanasghari

## Support Vector Machines (SVM): An Intuitive Explanation

Everything you always wanted to know about this powerful supervised ML algorithm

17 min read · Jul 1



732



4



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

Introduction

11 min read · Jun 22



35



See more recommendations