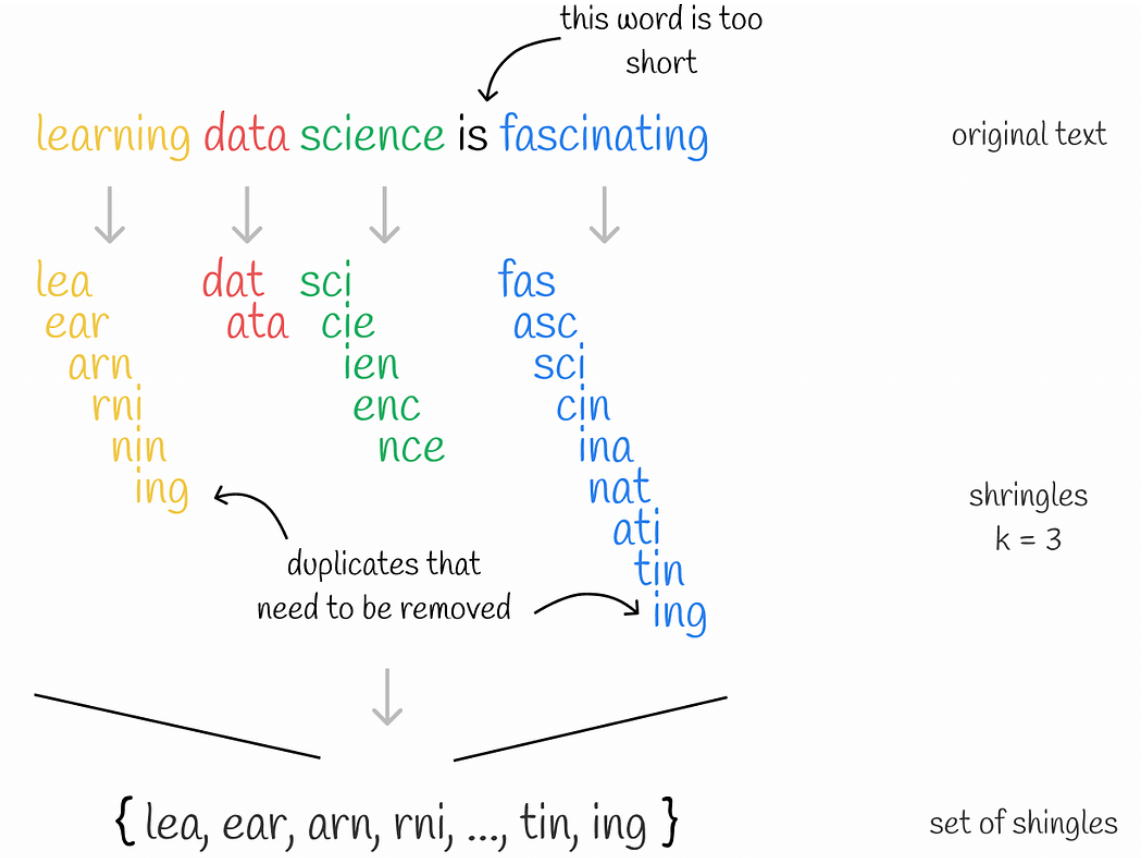
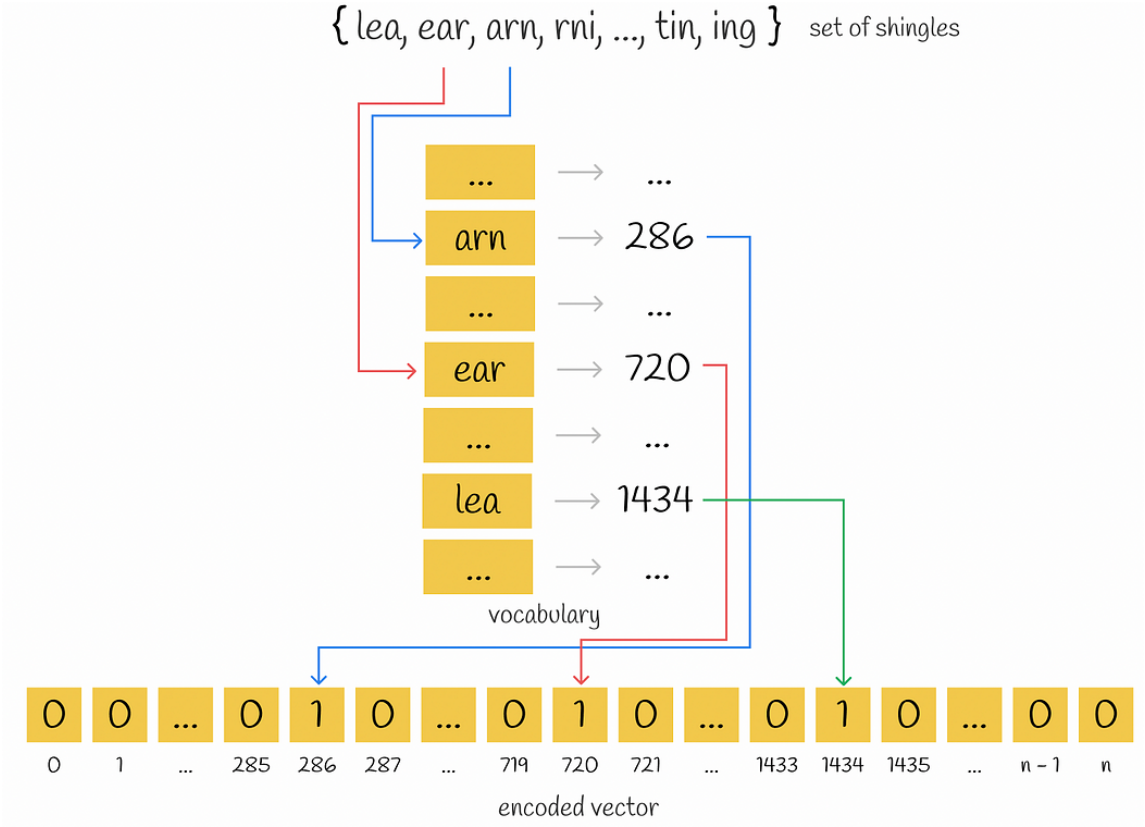
Shingling is the process of collecting *k*-grams on given texts. ***k*-gram** is a group of *k* sequential tokens. Depending on the context, tokens can be words or symbols. The ultimate goal of shingling is by using collected *k*-grams to encode each document. We will be using one-hot encoding for this. Nevertheless, other encoding methods can also be applied.



Collecting unique shringles of length k = 3 for the sentence “learning data science is fascinating”

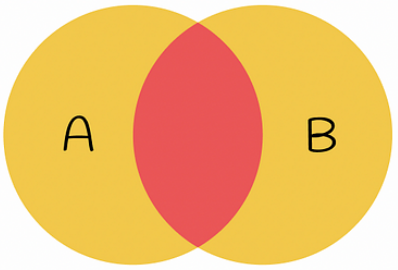
Firstly, unique *k*-grams for each document are collected. Secondly, to encode each document, a vocabulary is needed which represents a set of unique *k*-grams in all documents. Then for each document, a vector of zeros with the length equal to the size of the vocabulary is created. For every appearing k-gram in the document, its position in the vocabulary is identified and a *“1”* is placed at the respective position of the document vector. Even if the same *k*-gram appears several times in a document, it does not matter: the value in the vector will always be 1.



One-hot encoding

MinHashing

At this stage, initial texts have been vectorised. The similarity of vectors can be compared via **Jaccard index**. Remember that Jaccard index of two sets is defined as the number of common elements in both sets divided by the length of all the elements.



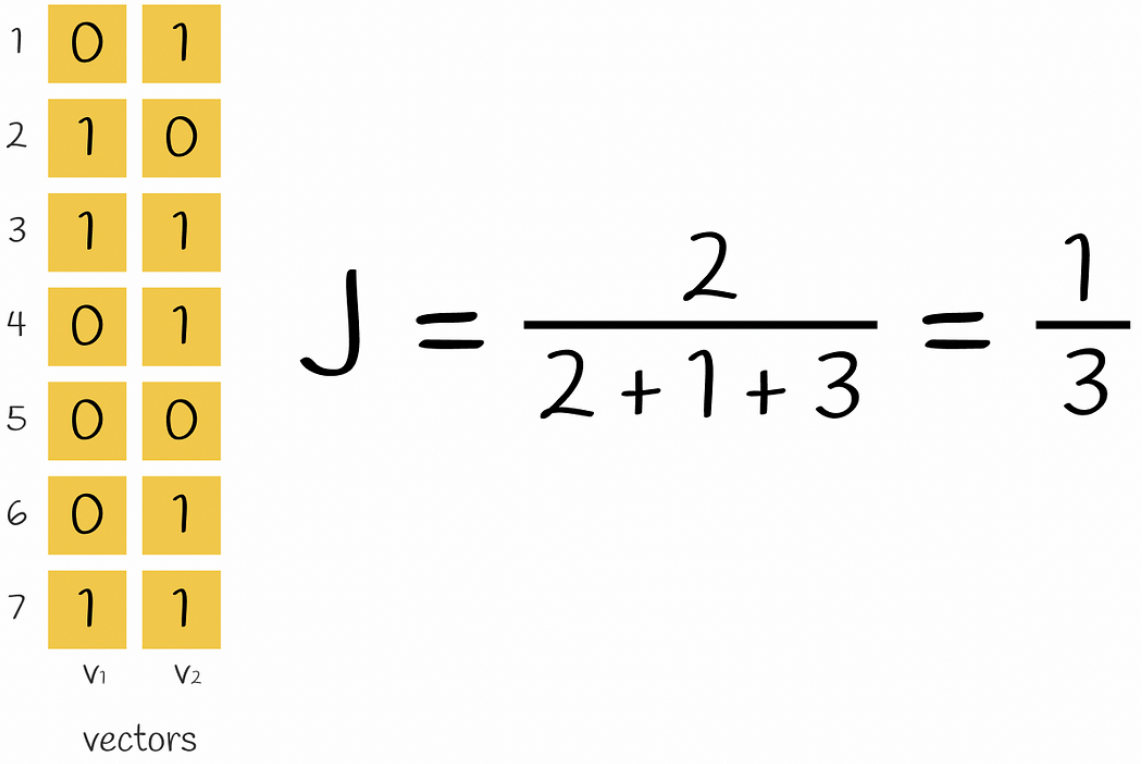
$$J = \frac{|A \cap B|}{|A \cup B|}$$

Jaccard Index is defined as the intersection over the union of two sets

If a pair of encoded vectors is taken, the intersection in the formula for Jaccard index is the number of rows that both contain 1 (i.e. *k*-gram appears in both vectors) and the union is the number of rows with at least one 1 (*k*-gram is presented at least in one of the vectors).

$$J = \frac{count(1\ 1)}{count(0\ 1) + count(1\ 0) + count(1\ 1)}$$

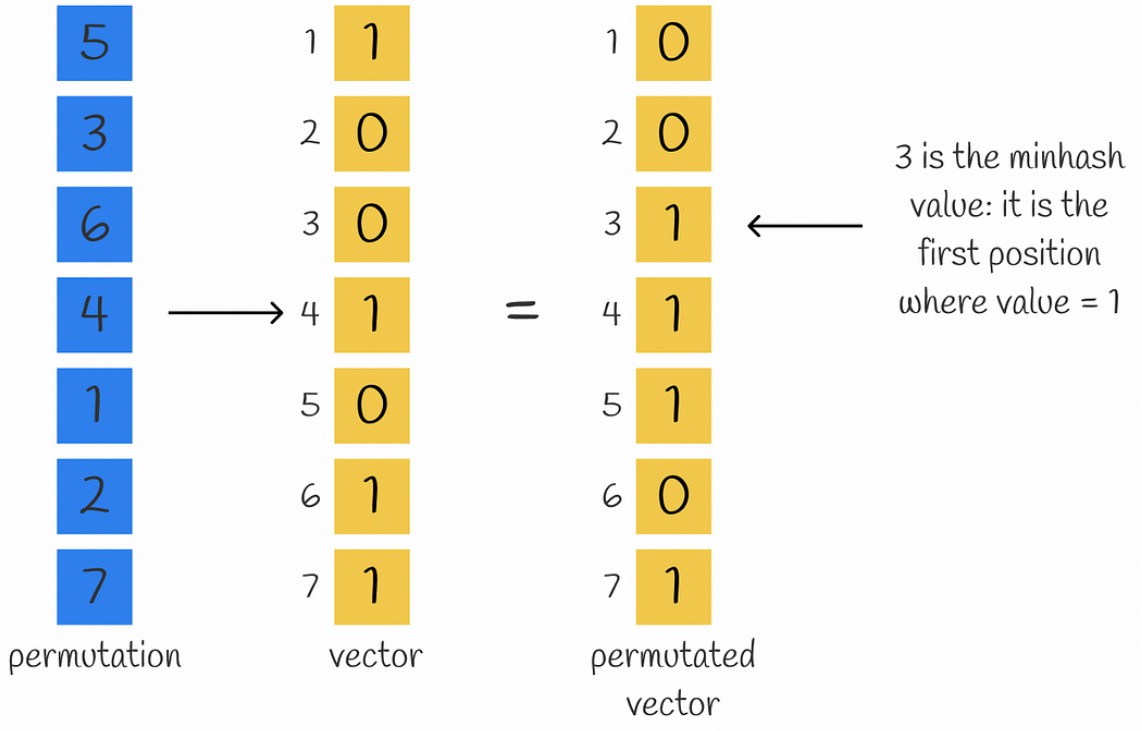
Formula for Jaccard Index of two vectors



Example of calculating Jaccard Index for two vectors using the formula above

The current problem right now is the sparsity of encoded vectors. Computing a similarity score between two one-hot encoded vectors would take a lot of time. Transforming them to a dense format would make it more efficient to operate on them later. Ultimately, the goal is to design such a function that will transform these vectors to a smaller dimension preserving the information about their similarity. The method that constructs such a function is called MinHashing.

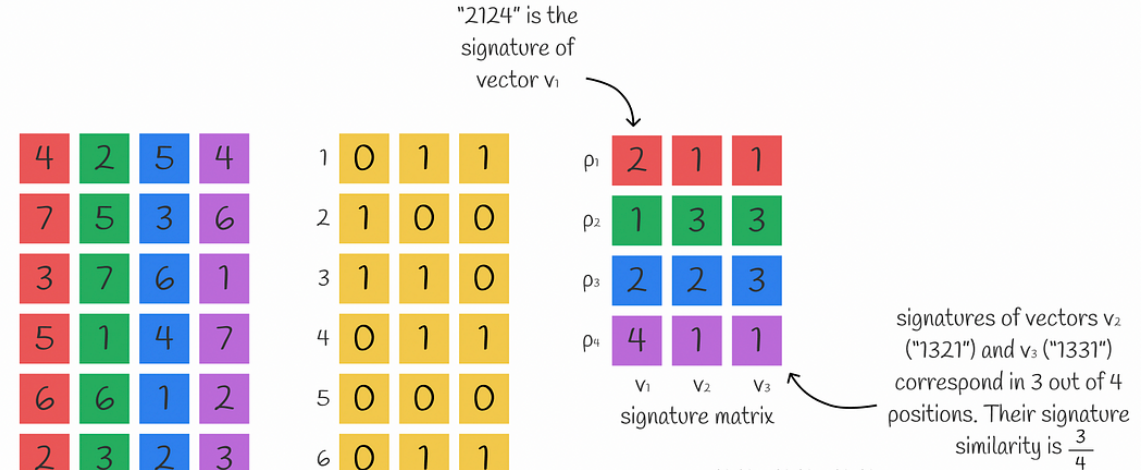
> **MinHashing** is a hash function that permutes the components of an input vector and then returns the first index where the permutated vector component equals 1.

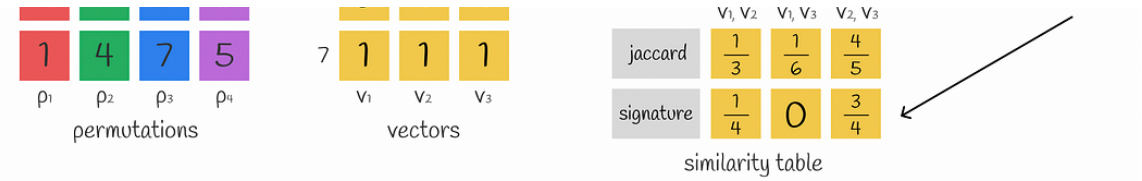


Example of calculating a minhash value for a given vector and permutation

For getting a dense representation of a vector consisting of *n* numbers, *n* minhash functions can be used to obtain *n* minhash values which form a **signature**.

It may not sound obvious at first but several minhash values can be used to approximate Jaccard similarity between vectors. In fact, the more minhash values are used, the more accurate the approximation is.



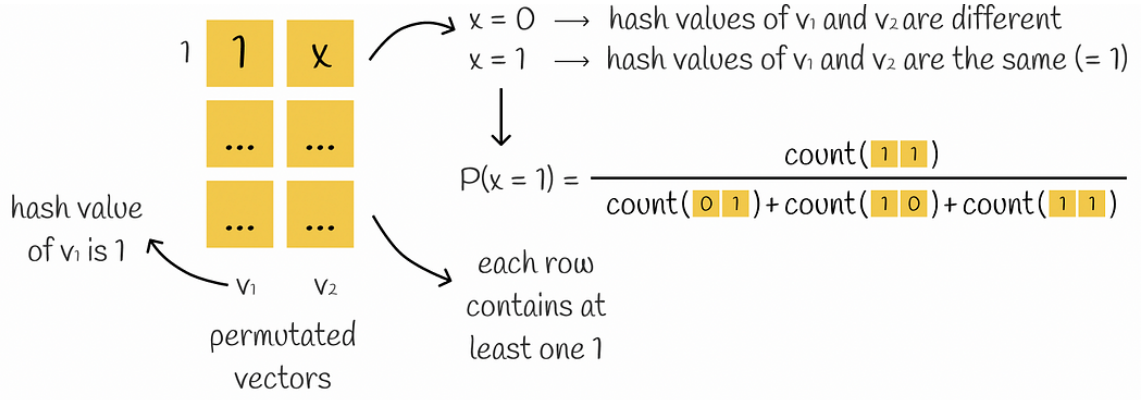


Calculation of signature matrix and how it is used to compute similarities between vectors. Similarities computed using Jaccard similarity and signatures should normally be approximately equal.

This is just a useful observation. It turns out that there is a whole theorem behind the scenes. Let us understand why Jaccard index can be calculated by using signatures.

Statement proof

Let us assume that a given pair of vectors contains only rows of type *01*, *10* and *11*. Then a random permutation on these vectors is performed. Since there exists at least one 1 in all the rows, then while computing both hash values, at least one of these two hash-value computation processes will stop at the first row of a vector with the corresponding hash value equal to 1.

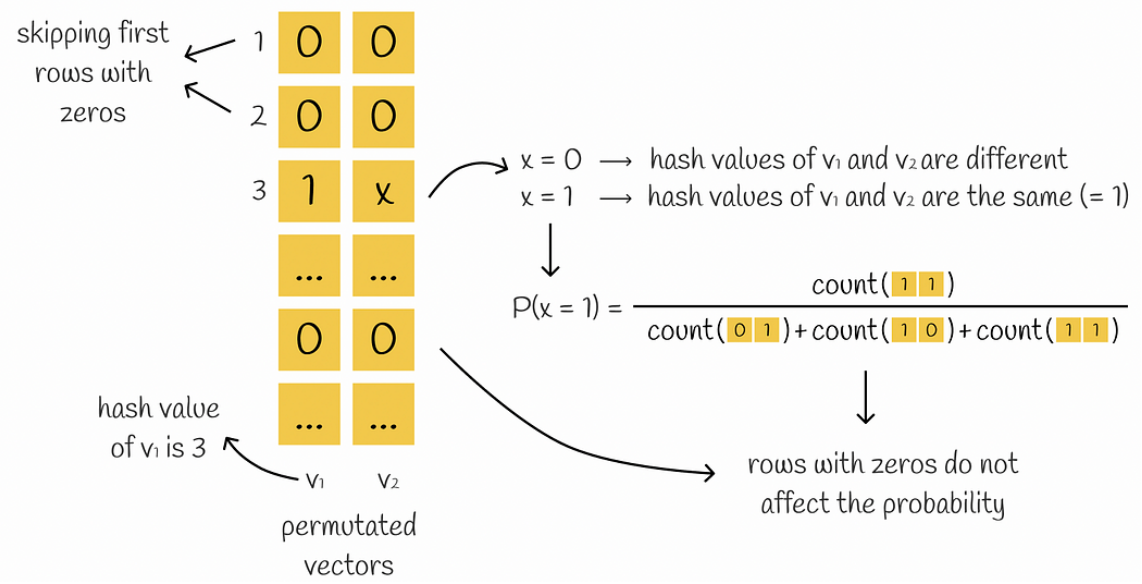


What is the probability that the second hash value will be equal to the first one? Obviously, this will only happen if the second hash value is also equal to 1. This means that the first row has to be of type *11*. Since the permutation was taken randomly, the probability of such an event is equal to *P = count(11) / (count(01) + count(10) + count(11)*). This expression is

absolutely the same as the Jaccard index formula. Therefore:

> **The probability of getting equal hash values for two binary vectors based on a random rows permutation equals the Jaccard index**.

However, by proving the statement above, we assumed that initial vectors did not contain rows of type *00*. It is clear that rows of type *00* do not change the value of Jaccard index. Similarly, the probability of getting the same hash values with rows of type *00* included does not affect it. For example, if the first permutated row is 00, then minhash algorithm just ignores it and switches to the next row until there exists at least one 1 in a row. **Of course, rows of type 00 can result in different hash values than without them but the probability of getting the same hash values stays the same**.



We have proven an important statement. But how the probability of getting the same minhash values can be estimated? Definitely, it is possible to generate all possible permutations for vectors and then calculate all minhash values to find the desired probability. For obvious reasons, this is not efficient because the number of possible permutations for a vector of size *n* equals *n!*. Nevertheless, the probability can be evaluated approximately:

let us just use many hash functions to generate that many hash values.

> The Jaccard index of two binary vectors approximately equals the number of corresponding values in their signatures.

$$P[(hash_\pi(v_i) = hash_\pi(v_j)] = jaccard\ (v_i, v_j)$$

Mathematical notation

It is easy to notice that taking longer signatures results in more accurate calculations.

LSH Function

At the current moment, we can transform raw texts into dense signatures of equal length preserving the information about similarity. Nevertheless, in practice, such dense signatures still usually have high dimensions and it would be inefficient to directly compare them.

Consider $n = 10^6$ documents with their signatures of length 100. Assuming that a single number of a signature requires 4 bytes to store, then the whole signature would require 400 bytes. For storing $n = 10^6$ documents, 400 MB of space is needed which is doable in reality. But comparing each document with each other in a brute-force manner would require approximately $5 * 10^{11}$ comparisons which is too much, especially when n is even larger.

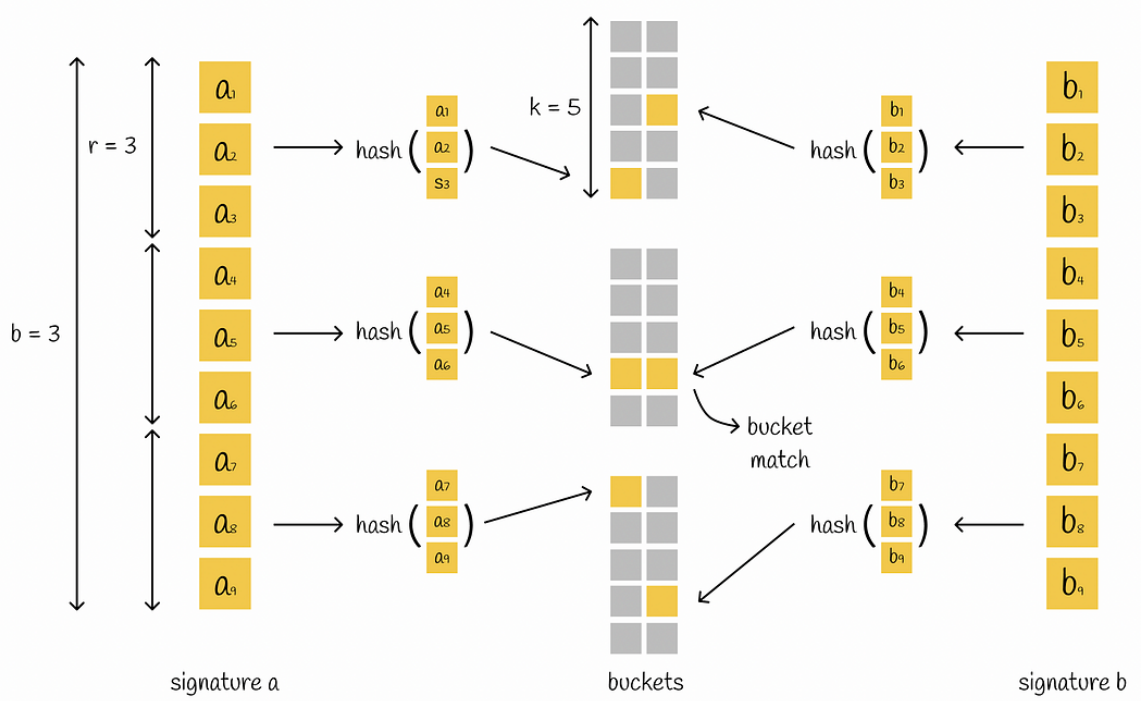
$$comparisons = \binom{n}{2} = \frac{n * (n-1)}{2}$$

To avoid the problem, it is possible to build a hash table to accelerate search performance but even if two signatures are very similar and differ only in 1 position, they are still likely to

have a different hash (because vector remainders are likely to be different). However, we normally want them to fall into the same bucket. This is where LSH comes to the rescue.

> **LSH** mechanism builds a hash table consisting of several parts which puts a pair of signatures into the same bucket if they have at least one corresponding part.

LSH takes a signature matrix and horizontally divides it into equal *b* parts called **bands** each containing *r* **rows**. Instead of plugging the whole signature into a single hash function, the signature is divided by *b* parts and each subsignature is processed independently by a hash function. As a consequence, each of the subsignatures falls into separate buckets.



Example of using LSH. Two signatures of length 9 are divided into b = 3 bands each containing r = 3 rows. Each subvector is hashed into one of k possible buckets. Since there is a match in the second band (both subvectors have the same hash value), we consider a pair of these signatures as candidates to be the nearest neighbours.

If there is at least one collision between corresponding subvectors of two different signatures, the signatures are considered candidates. As we can see, this condition is more flexible since for considering vectors as candidates they do not need to be absolutely equal. Nevertheless, this increases the number of false positives: a pair of different signatures can have a single corresponding part but in overall be completely different. Depending on the problem, it is always better to optimize parameters *b*, *r* and *k*.

Error rate

With LSH, it is possible to estimate the probability that two signatures with similarity *s* will be considered as candidates given a number of bands *b* and number of rows *r* in each band. Let us find the formula for it in several steps.

$$P = s$$

The probability that one random row of both signatures is equal

$$P = s^r$$

The probability that one random band with r rows is equal

$$P = 1 - s^r$$

The probability that one random band with r rows is different

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

The probability that all b bands in the table are different

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

The probability that at least one of b bands is equal, so two signatures are candidates

Note that the formula does not take into consideration collisions when different subvectors accidentally hash into the same bucket. Because of this, the real probability of signatures being the candidates might insignificantly differ.

Example

For getting a better sense of the formula we have just obtained, let us consider a simple example. Consider two signatures with the length of 35 symbols which are equally split into 5 bands with 7 rows each. Here is the table which represents the probability of having at least one equal band based on their Jaccard similarity:

s, %	0	10	20	30	40	50	60	70	80	90	100
P, %	0	0.007	0.224	1.69	6.95	19.9	43.3	72.4	93.8	99.8	100

Probability P of getting at least one corresponding band of two signatures based on their similarity s

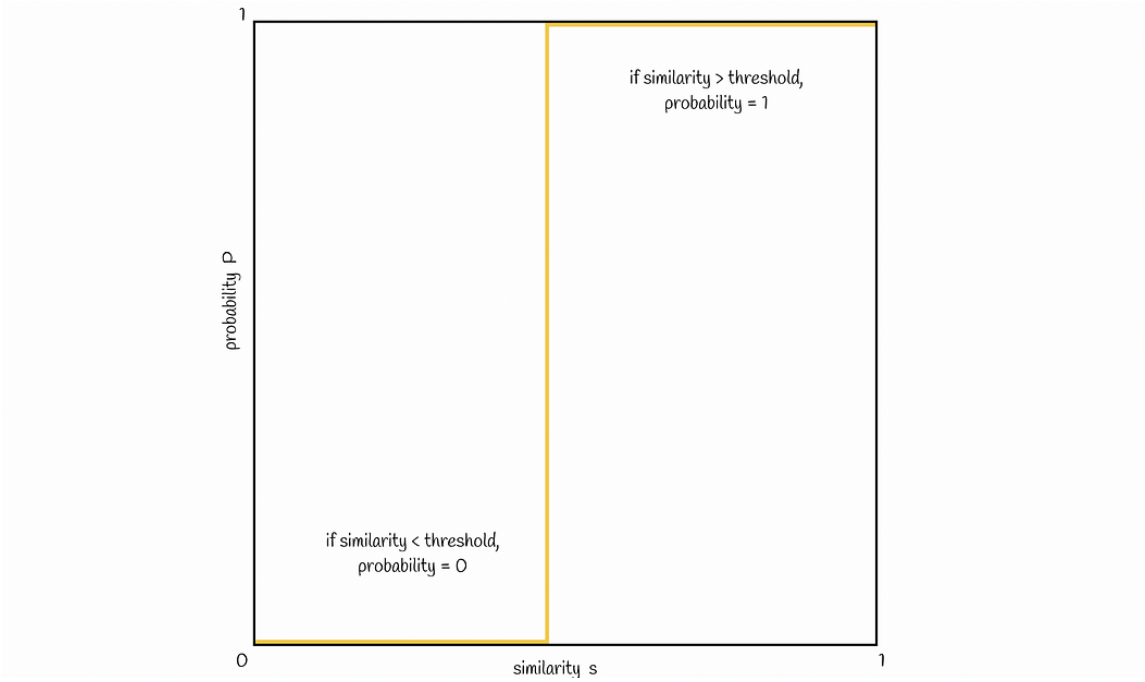
We notice that if two similar signatures have the Jaccard similarity of 80%, then they have a corresponding band in 93.8% of cases (*true positives*). In the rest 6.2% of scenarios such a pair of signatures is *false negative*.

Now let us take two different signatures. For instance, they are similar only by 20%. Therefore, they are *false positive* candidates in 0.224% of cases. In other 99.776% of cases, they do not have a similar band, so they are *true negatives*.

Visualisation

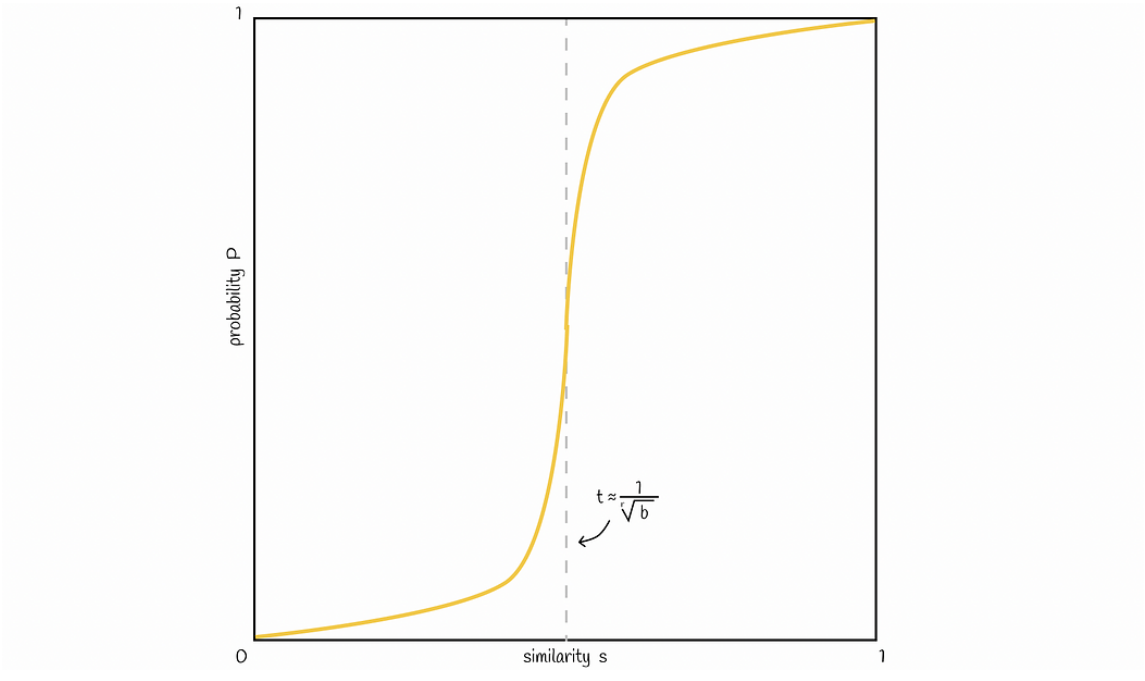
Let us now visualise the connection between similarity *s* and probability *P* of two signatures becoming candidates. Normally with higher signature similarity *s,* signatures should have a higher probability of being candidates. Ideally, it would look like

below:



Ideal scenario. A pair of signatures is considered candidates only if their similarity is greater than a certain threshold t

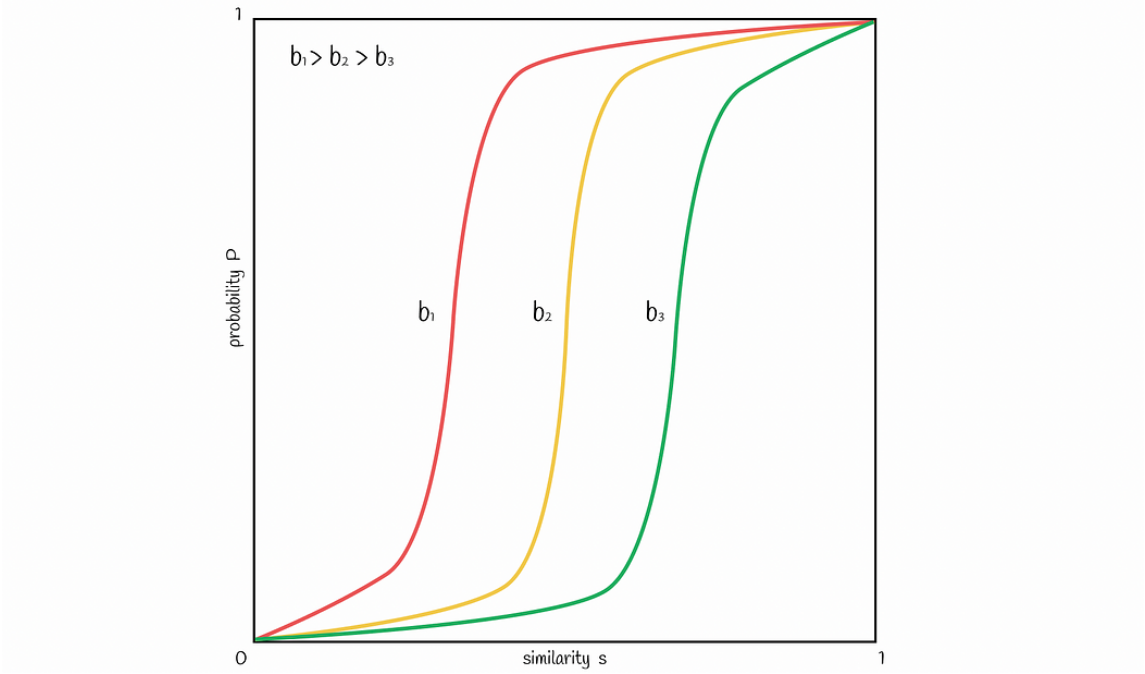
Based on the probability formula obtained above, a typical line would look like in the figure below:



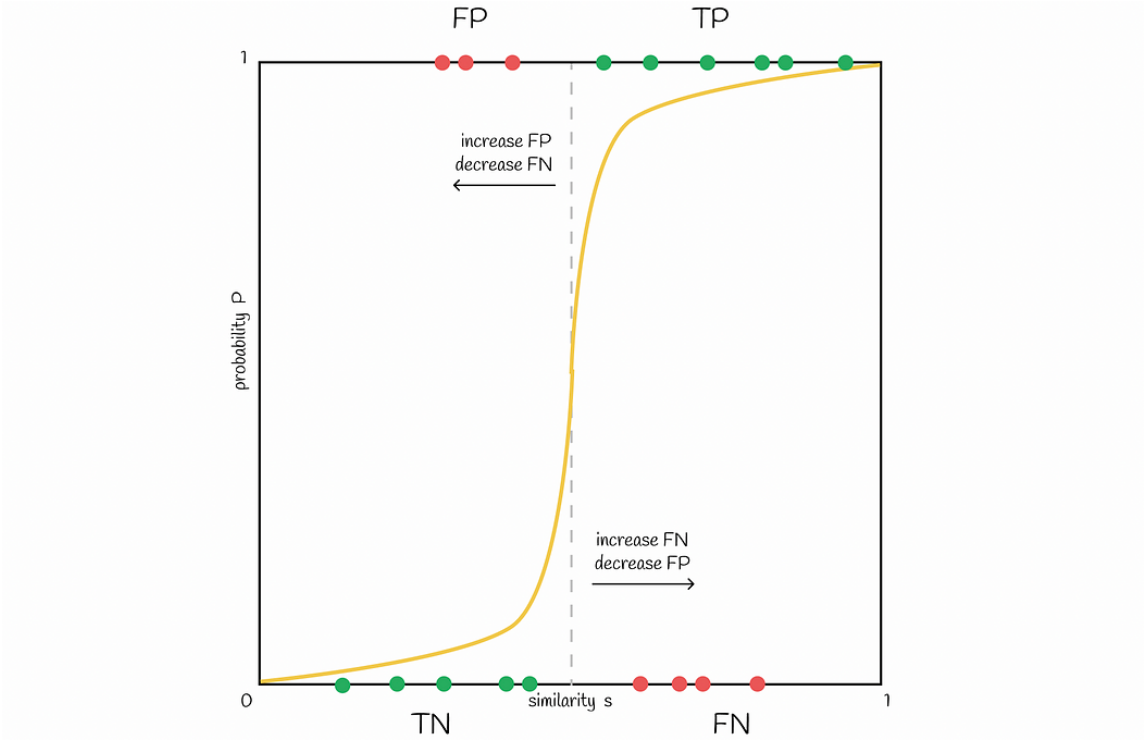
A typical line that slowly increases in the beginning and at the end and has a steep slope at a threshold t given by the approximate probability formula in the figure

It is possible to vary the number of bands *b* to shift the line in the

figure to the left or to the right. Increasing *b* moves the line to the left and results in more *FP*, decreasing — shifts it to the right and leads to more *FN*. It is important to find a good balance, depending on the problem.



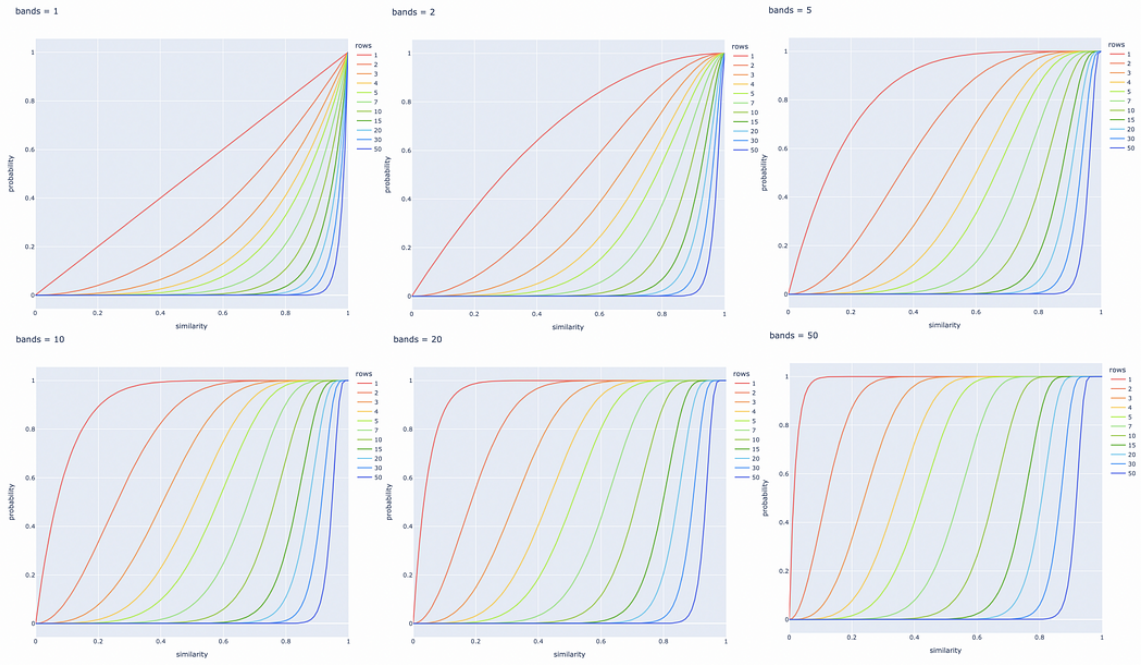
With a higher number of bands the line moves to the left, with lower — to the right



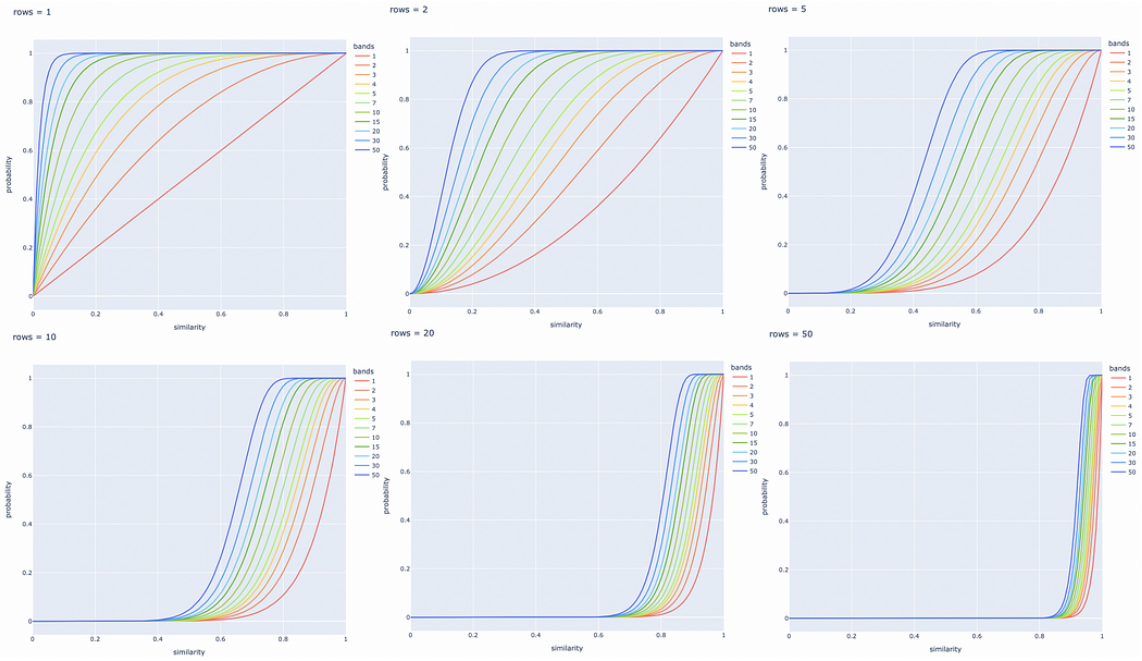
Moving the threshold to the left increases FP while shifting it to the right increases FN

Experimentations with different numbers of bands and rows

Several line plots are built below for different values of *b* and *r*. It is always better to adjust these parameters based on the specific task to successfully retrieve all pairs of similar documents and ignore those with different signatures.



Adjusting number of bands



Adjusting number of rows

Conclusion