**This member-only story is on us.** Upgrade to access all of Medium.

Member-only story

# Semantic Search: Measuring Meaning From Jaccard to Bert

Supercharge search with these stellar technologies

James Briggs · Follow

Published in Towards Data Science · 11 min read · Jun 29, 2021

262 1

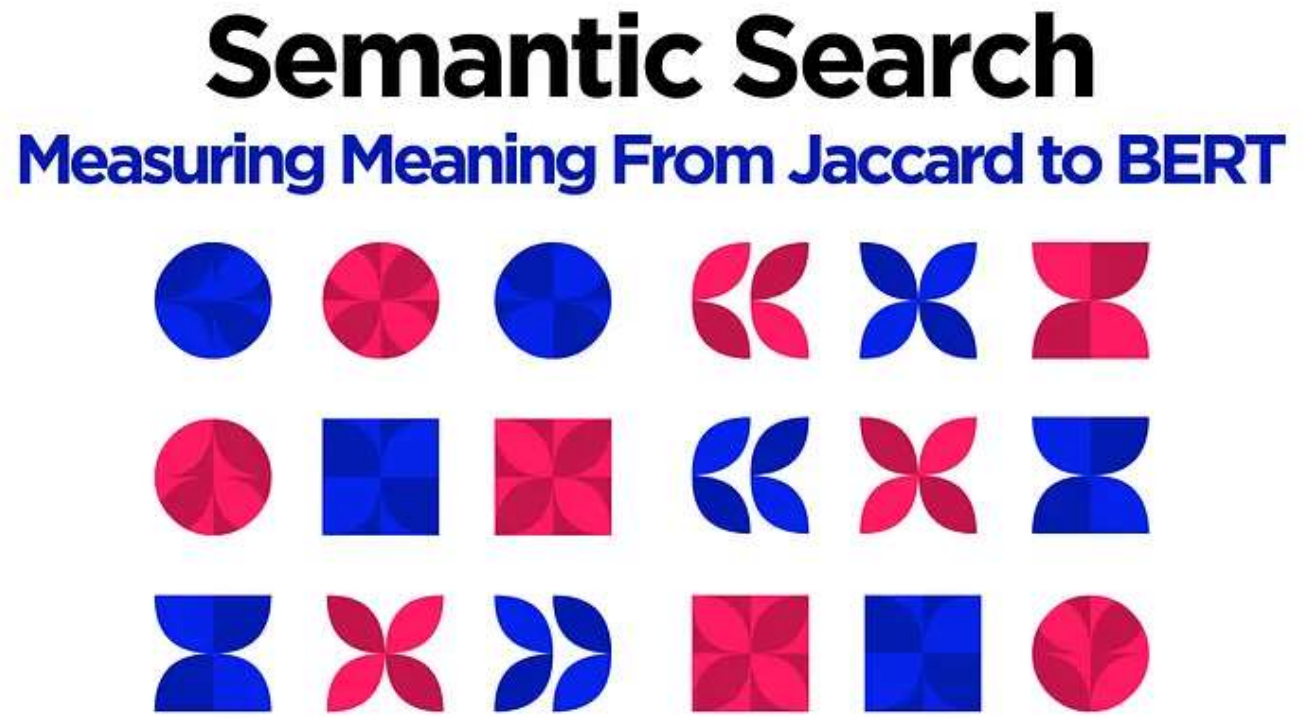



Image by author — original article on Pinecone.io

Search Medium Write

pieces of information together.

There's a strong chance that you found this article through a search engine — most likely Google. Maybe you searched something like "what is similarity search?" or "traditional vs vector similarity search".

Google processed your query and used many of the same similarity search essentials that we will learn about in this article, to bring you to — this article.

If similarity search is at the heart of the success of a $1.65T company — the world's fifth most valuable company in the world [1], there's *a good chance* it's worth learning more about.

Similarity search is a complex topic and there are countless techniques for building effective search engines.

In this article, we'll cover a few of the most interesting — and powerful — of these techniques — focusing specifically on semantic search. We'll learn how they work, what they're good at, and how we can implement them ourselves.

## Traditional Search

We start our journey down the road of search in the traditional camp, here we find a few key players like:

- **Jaccard Similarity**
- **w-shingling**
- Pearson Similarity
- **Levenshtein distance**
- Normalized Google Distance

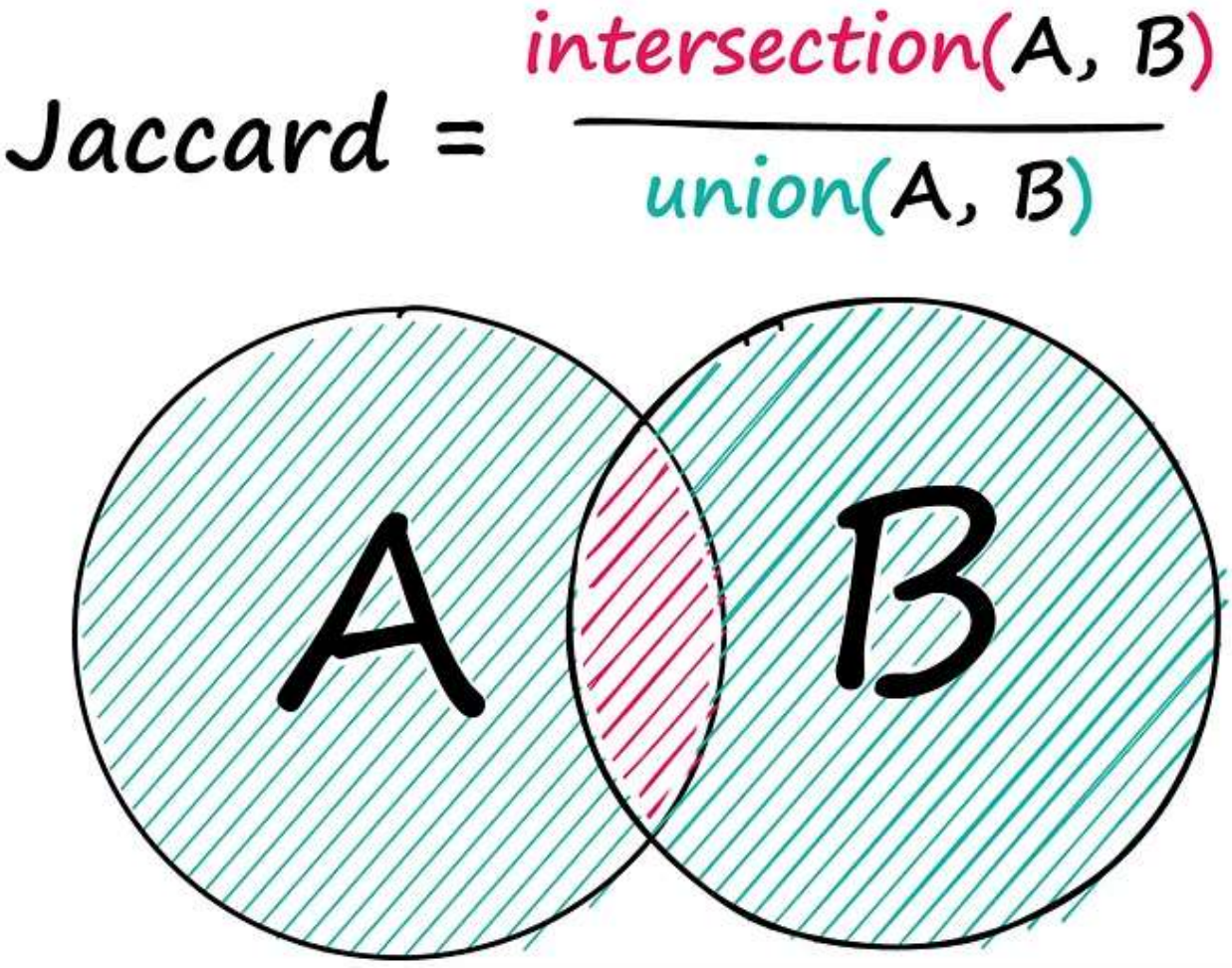
All are great metrics to use with similarity search — of which we'll cover three of the most popular, Jaccard similarity, w-shingling, and Levenshtein distance.

Video walkthrough covering the same three traditional similarity methods.

## Jaccard Similarity

Jaccard similarity is a simple, but sometimes powerful similarity metric. Given two sequences, **A** and **B** — we find the number of shared elements

between both and divide this by the total number of elements from both sequences.

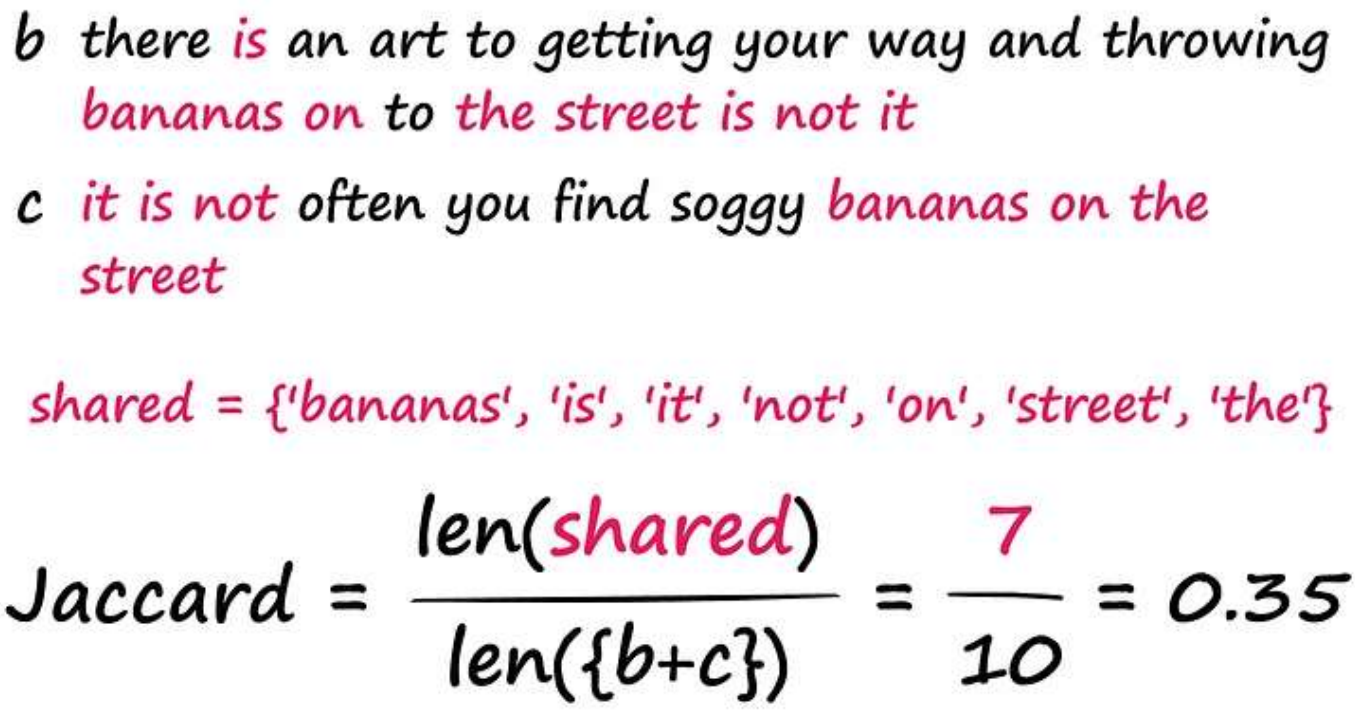



Jaccard similarity measures the intersection between two sequences over the union between the two sequences.

Given two sequences of integers, we would write:

Here we identified **two** shared *unique* integers, `3` and `4` — between two sequences with a total of ten integers in both, of which **eight** are unique values — `2/8` gives us our Jaccard similarity score of `0.25`.

We could perform the same operation for text data too, all we do is replace *integers* with *tokens*.




Jaccard similarity calculated between two sentences **a** and **b.**

We find that sentences `b` and `c` score much better, as we would expect. Now, it isn't perfect — two sentences that share nothing but words like *'the'*, *'a'*, *'is'*, etc — could return high Jaccard scores despite being semantically dissimilar.

These shortcomings can be solved partially using preprocessing techniques like stopword removal, stemming/lemmatization, and so on. However, as we'll see soon — some methods avoid these problems altogether.

### w-Shingling

Another similar technique is **w-shingling.** w-shingling uses the exact same logic of *intersection / union* — but with 'shingles'. A 2-shingle of sentence **a** would look something like:

```
a = {'his thought', 'thought process', 'process is', ...}
```

We would then use the same calculation of `intersection / union` between our *shingled* sentences like so:

Using a 2-shingle, we find three matching shingles between sentences **b** and **c**, resulting in a similarity of **0.125**.

### Levenshtein Distance

Another popular metric for comparing two strings is the Levenshtein distance. It is calculated as the number of operations required to change one string into another — and it's calculated with:

$$lev(a,b) = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0 \\ 1_{(a_i \neq b_i)} + \min \begin{cases} lev(a_{i-1}, b_j) \\ lev(a_i, b_{j-1}) \\ lev(a_{i-1}, b_{j-1}) \end{cases} & \text{else} \end{cases}$$

Levenshtein distance formula.

Now, this is a pretty complicated-looking formula — if you understand it, great! If not, don't worry — we'll break it down.

The variables `a` and `b` represent our two strings, `i` and `j` represent the character position in `a` and `b` respectively. So given the strings:

$a$ = Levenshtein $b$ = Livinshten

'Levenshtein' and a mispelling 'Livinshten'.

We would find:

$a$ = Levenshtein $b$ = Livinshten

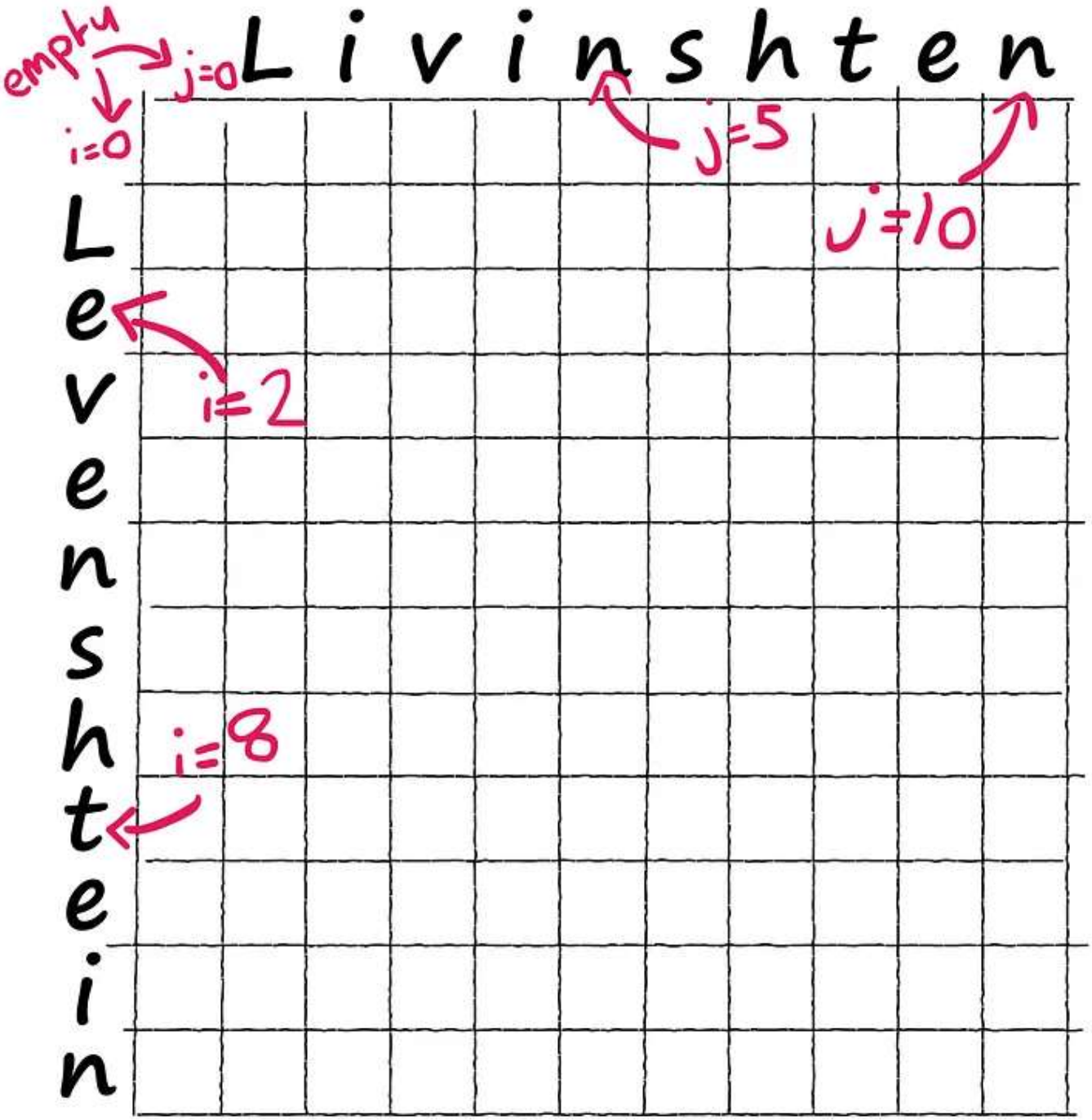
$a_{i=3}$ = v $b_{j=1}$ = L

$a_{i=7}$ = h $b_{j=8}$ = t

We index the word itself from 1 to the length of the word, the zeroth index does exist as a **none** character (more on that next).

Easy! Now, a great way to grasp the logic behind this formula is through visualizing the Wagner-Fischer algorithm — which uses a simple matrix to calculate our Levenshtein distance.

We take our two words `a` and `b` and place them on either axis of our matrix — we include our *none* character as an empty space.

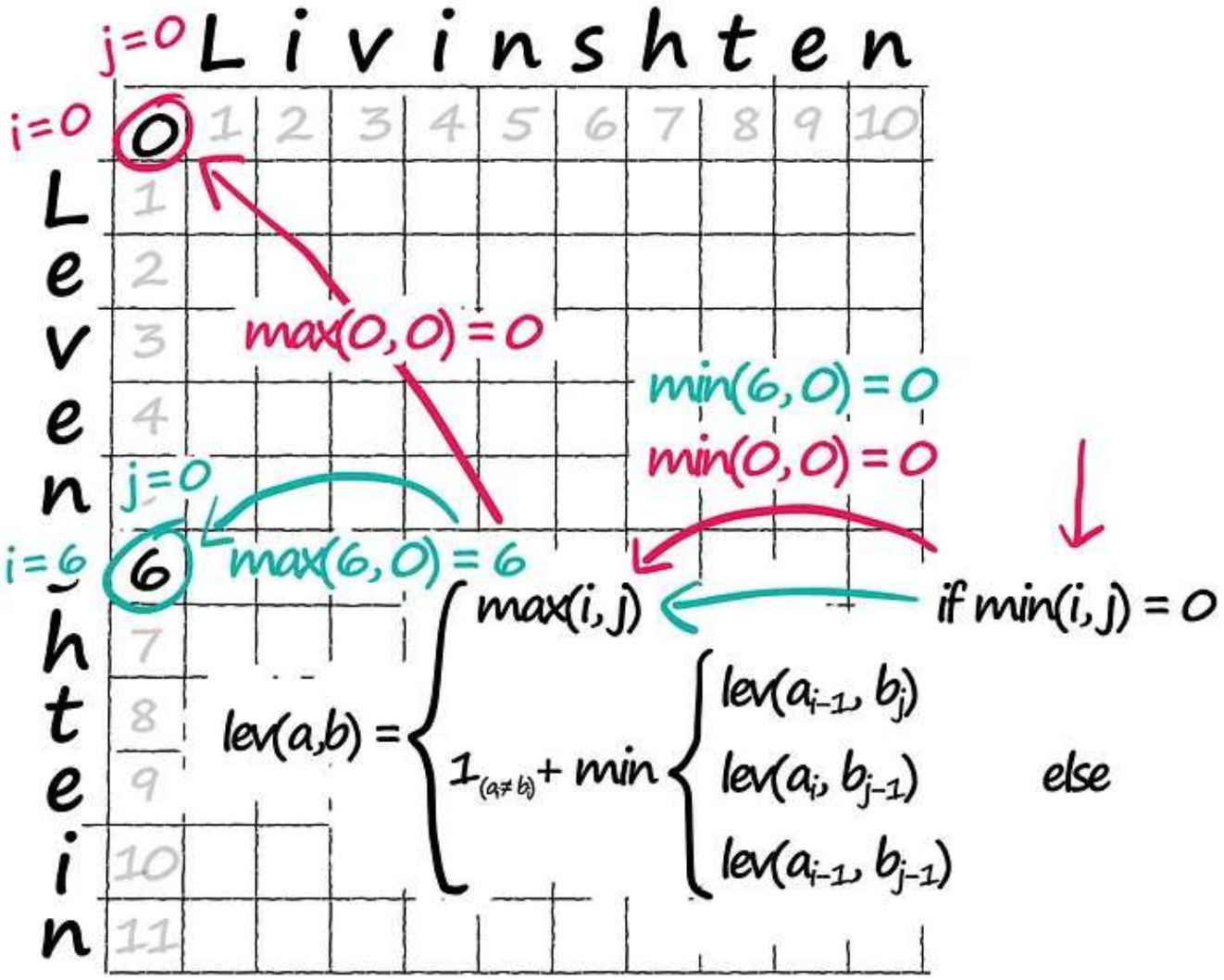


Our empty Wagner-Fischer matrix — we'll be using this to calculate the Levenshtein distance between **'Levenshtein'** and **'Livinshten'**.

Initializing our empty Wagner-Fischer matrix in code.

Then we iterate through every position in our matrix and apply that complicated formula we saw before.

The first step in our formulae is `if min(i, j) = 0` — all we're saying here is, out of our two positions `i` and `j`, are either `0`? If so, we move across to `max(i, j)` which tells us to assign the current position in our matrix the higher of the two positions `i` and `j`:



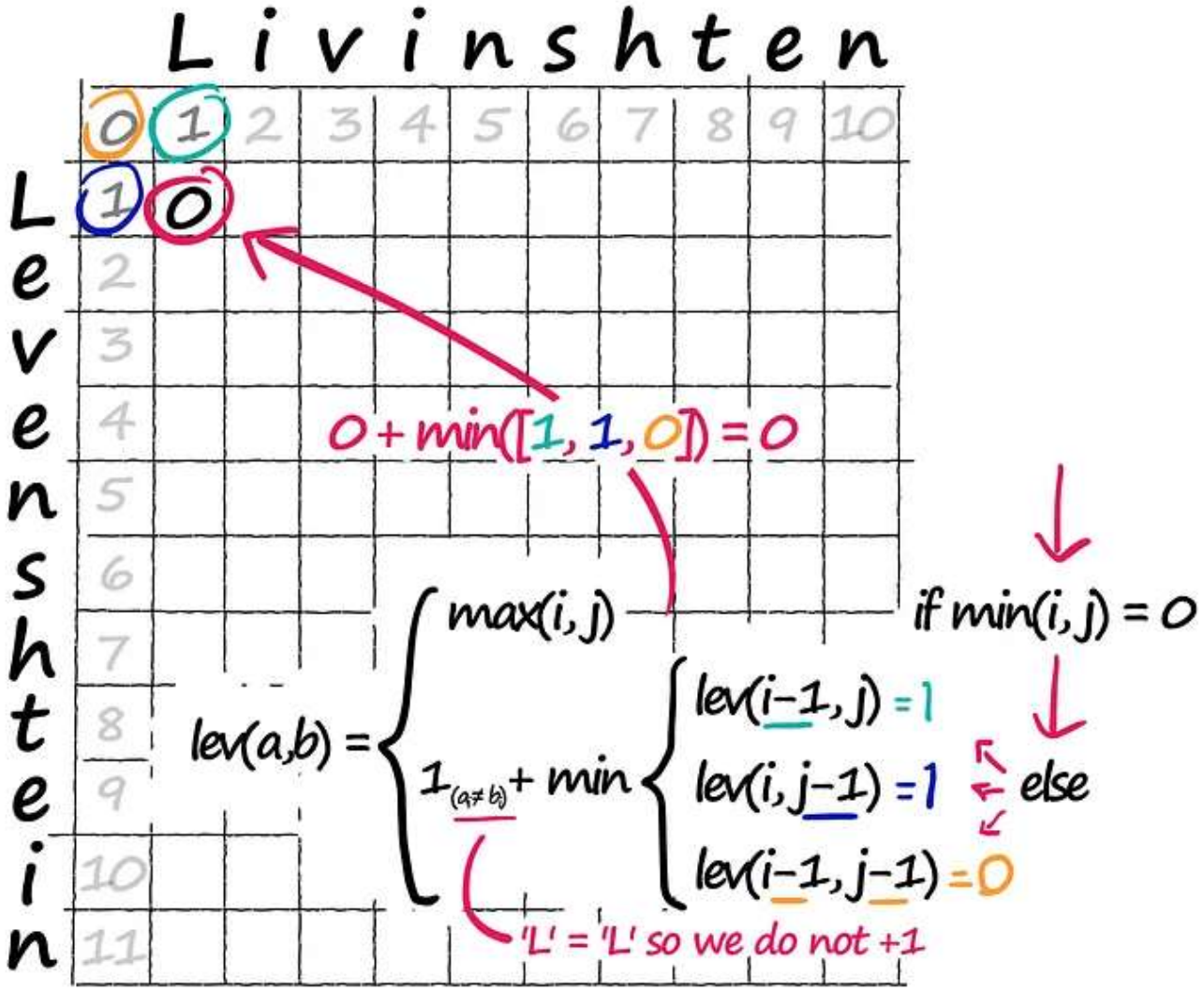

We start on the right, along the edges where **i** and/or **j** is 0, the matrix position will be populated with **max(i, j)**.

The min(i,j) == 0 followed by the max(i,j) operation visualized above — translated into code.

Now, we've dealt with the outer edges of our matrix — but we still need to calculate the inner values — which is where our optimal path will be found.

Back to `if min(i, j) = 0` — what if neither are `0`? Then we move onto that complex part of the equation inside the `min {` section. We need to calculate a value for each row, then we take the **min**imum value.

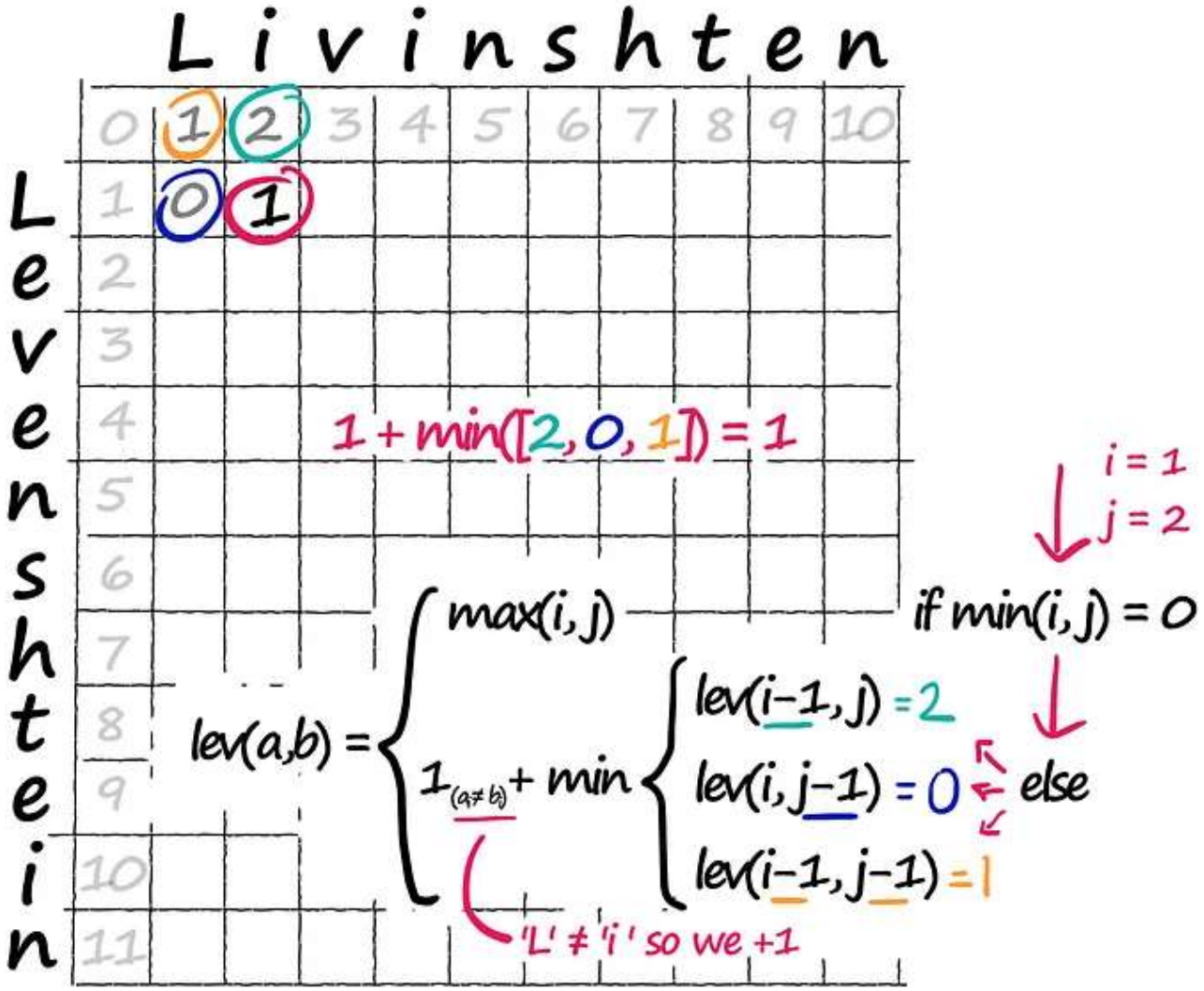
Now, we already know these values — they're in our matrix:




For each new position in our matrix, we take the minimum value from the three neighboring positions (circled — top-left).

`lev(i-1, j)` and the other operations are all indexing operations — where we extract the value in that position. We then take the minimum value of the three.

There is just one remaining operation. The `+1` on the left should only be applied if `a[i] != b[i]` — this is the penalty for mismatched characters.




If a[i] != b[j] we add **1** to our minimum value — this is the penalty for mismatched characters.

Placing all of this together into an iterative loop through the full matrix looks like this:

The full Levenshtein distance calculation using a Wagner-Fischer matrix.

We've now calculated each value in the matrix — these represent the number of operations required to convert from string `a` up to position `i` to string `b` up to position `j`.

We're looking for the number of operations to convert `a` to `b` — so we take the bottom-right value of our array at `lev[-1, -1]`.


The optimal path through our matrix — in position [-1, -1] at the bottom-right we have the Levenshtein distance between our two strings.

## Vector Similarity Search

For vector-based search, we typically find one of several vector building methods:

- TF-IDF

- **BM25**
- word2vec/doc2vec
- **BERT**
- USE

In tandem with some implementation of *approximate* nearest neighbors (ANN), these vector-based methods are the MVPs in the world of similarity search.

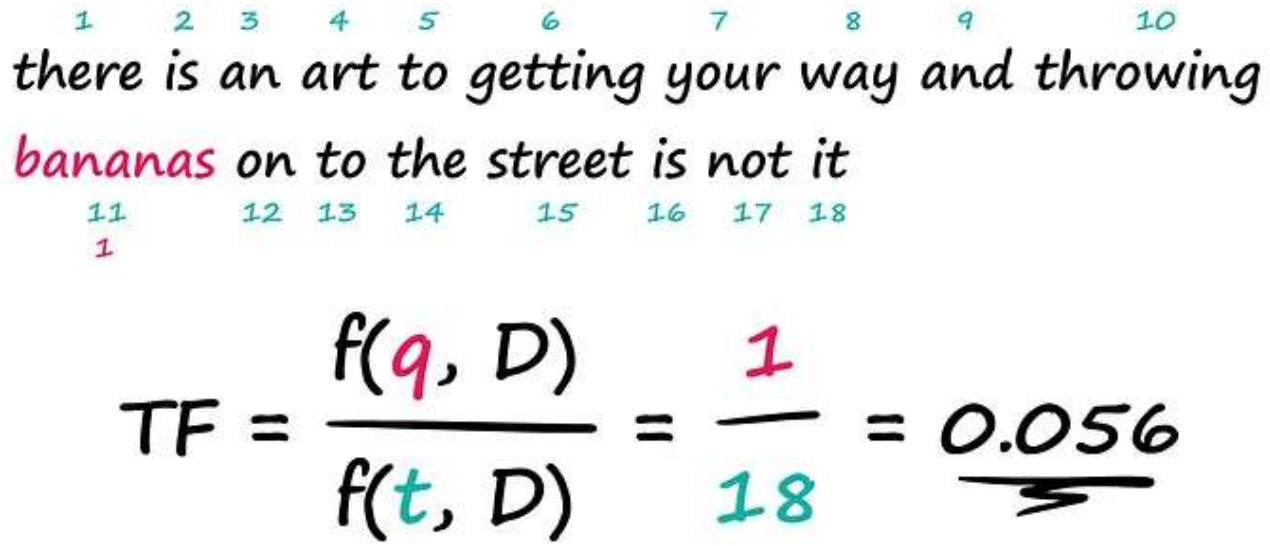
We'll cover TF-IDF, BM25, and BERT-based approaches — as these are easily the most common and cover both sparse and dense vector representations.

Video walkthrough covering the same three vector-based similarity methods.

1. **TF-IDF** — the respected grandfather of vector similarity search, born back in the 1970s. It consists of two parts, Term Frequency (TF) and

Inverse Document Frequency (IDF).

The TF component counts the number of times a term appears within a document and divides this by the total number of terms in that same document.



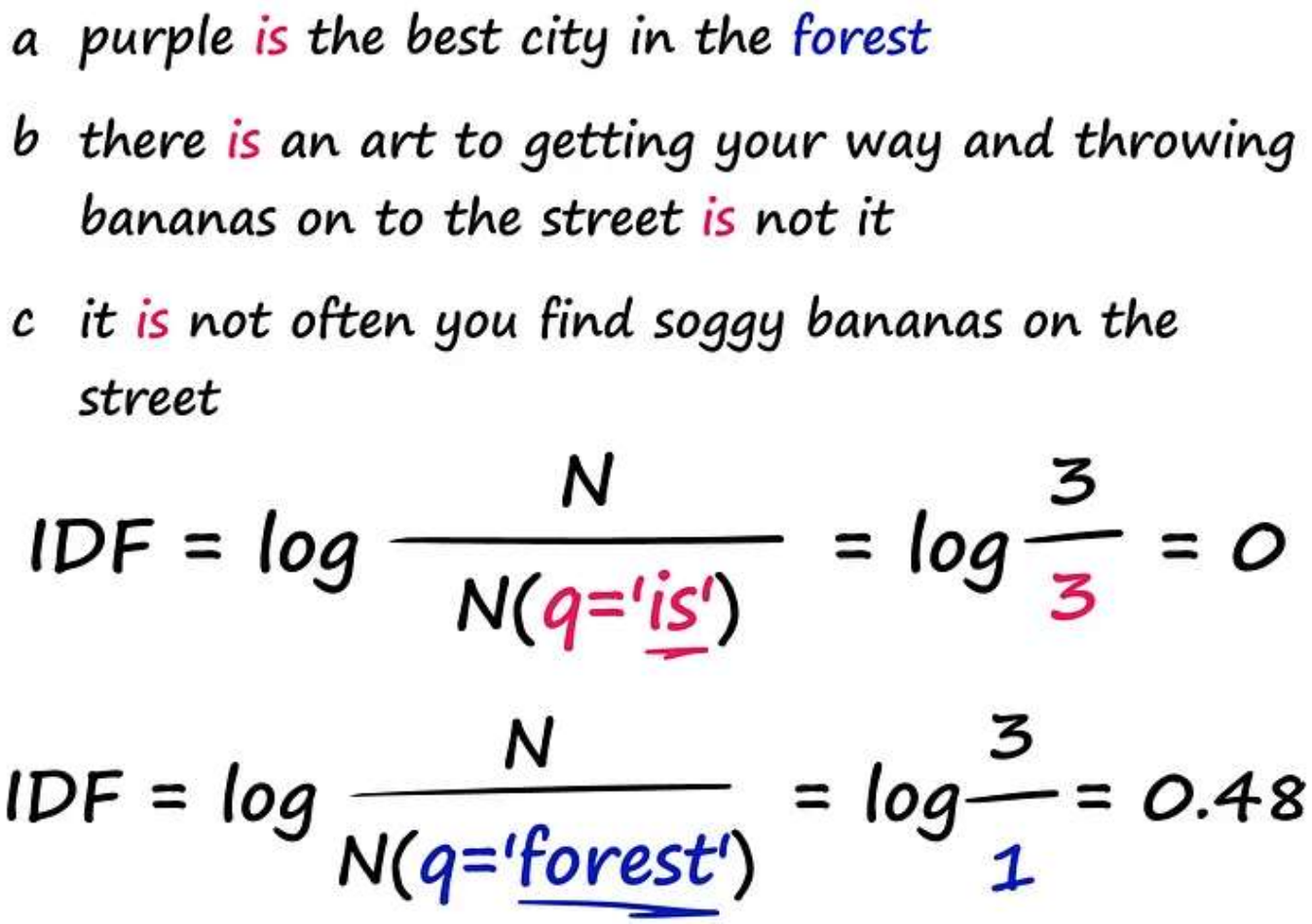

The term frequency (TF) component of TF-IDF counts the frequency of our query ('bananas') and divides by the frequency of all tokens.

That is the first half of our calculation, we have the **f**requency of our **q**uery within the current **D**ocument `f(q,D)` — over the **f**requency of all **t**erms within the current **D**ocument `f(t,D)`.

The **T**erm **F**requency is a good measure but doesn't allow us to differentiate between common and uncommon words. If we were to search for the word 'the' — using TF alone we'd assign this sentence the same relevance as had we searched 'bananas'.

That's fine until we begin comparing documents or searching with longer queries. We don't want words like *'the'*, *'is'*, or *'it'* to be ranked as highly as *'bananas'* or *'street'*.

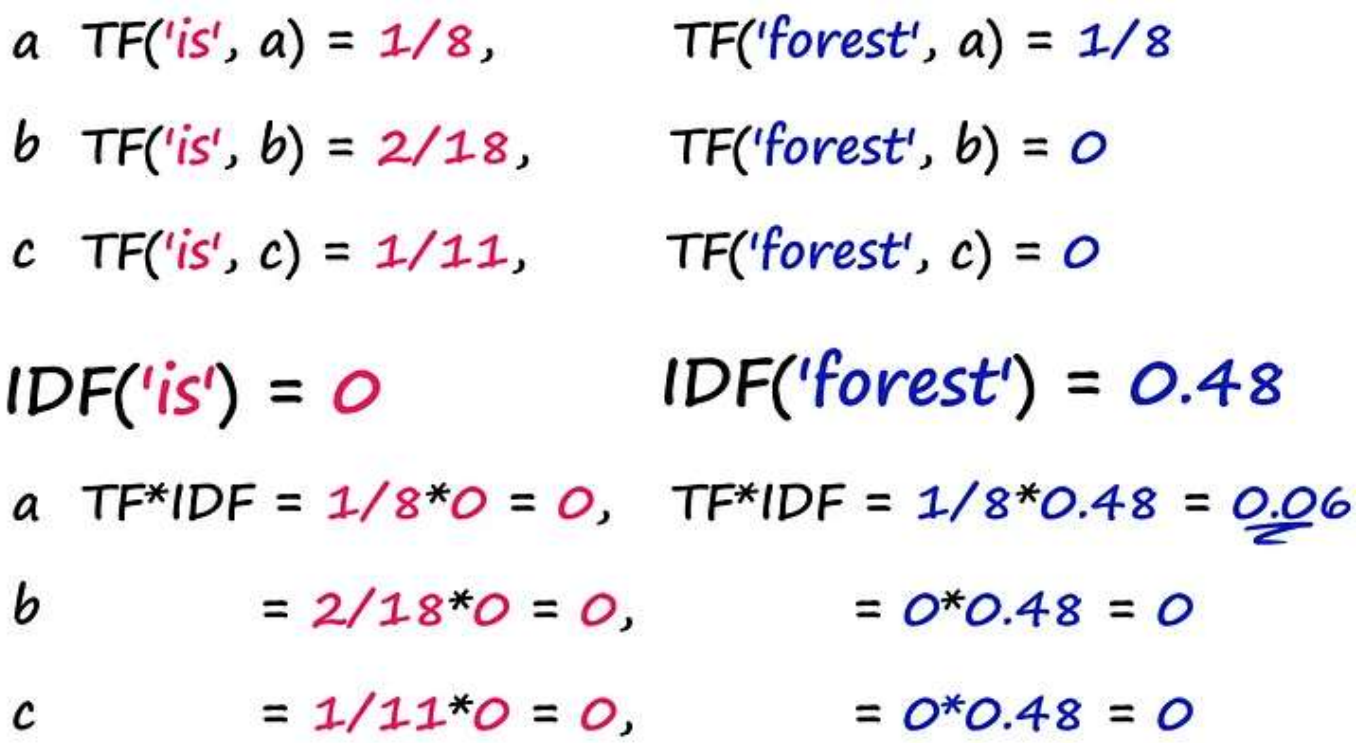
Ideally, we want matches between rarer words to score higher. To do this, we can multiply TF by the second term — IDF. The **I**nverse **D**ocument **F**requency measures how common a word is across *all* of our documents.




The inverse document frequency (IDF) component of TF-IDF counts the number of documents that contain our query.

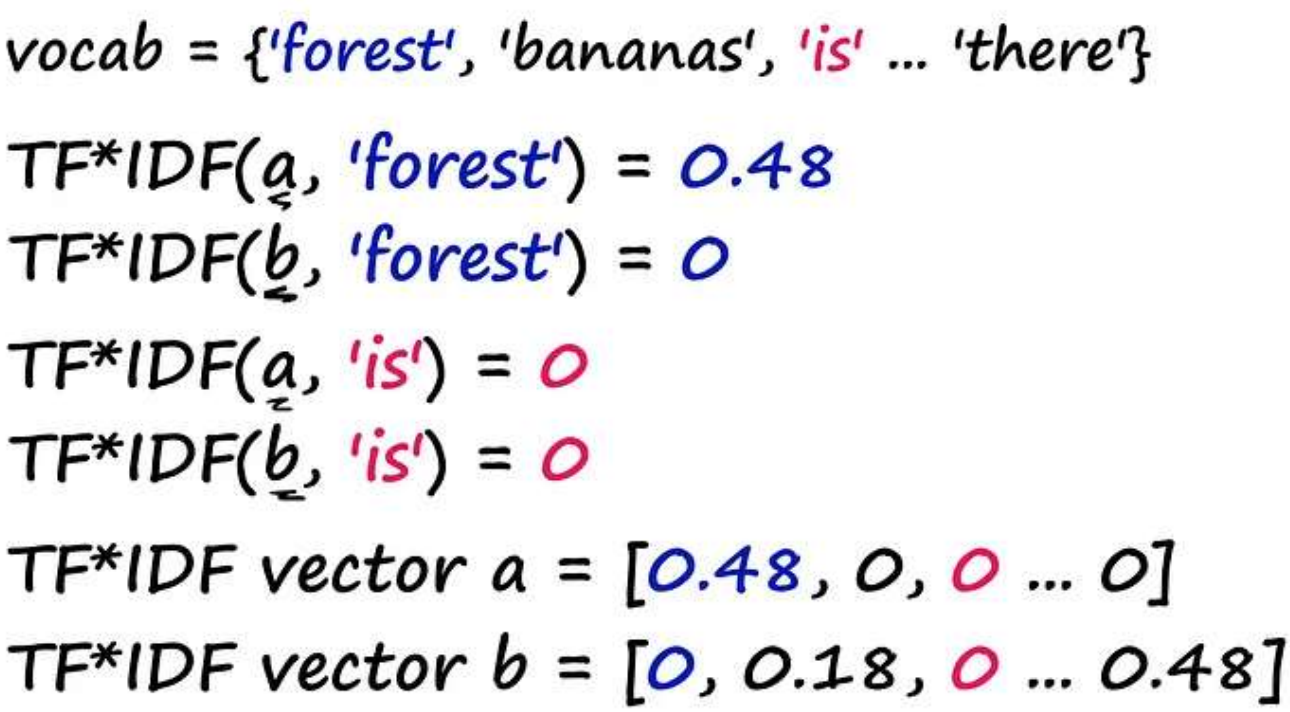
In this example, we have three sentences. When we calculate the IDF for our common word *'is'*, we return a much lower number than that for the rarer word *'forest'*.

If we were to then search for both words *'is'* and *'forest'* we would merge TF and IDF like so:




We calculate the **TF('is', D)** and **TF('forest', D)** scores for docs **a**, **b**, and **c.** The IDF value is across all docs — so we calculate just **IDF('is')** and **IDF('forest')** once. Then, we get TF-IDF values for both words in each doc by **multiplying** the **TF** and **IDF** components. Sentence **a** scores highest for **'forest'**, and **'is'** always scores **0** as the **IDF('is')** score is **0.**

That's great, but where does *vector* similarity search come into this? Well, we take our vocabulary (a big list of all words in our dataset) — and calculate the TF-IDF for each and every word.




We calculate the TF-IDF value for every word in our vocabulary to create a TF-IDF vector. This process is repeated for each document.

We can put all of this together to create our TF-IDF vectors like so:

From there we have our TF-IDF vector. It's worth noting that vocab sizes can easily be in the 20K+ range, so the vectors produced using this method are incredibly sparse — which means we cannot encode any semantic meaning.

2. **BM25** — the successor to TF-IDF, Okapi BM25 is the result of optimizing TF-IDF primarily to normalize results based on document length.

TF-IDF is great but can return questionable results when we begin comparing several mentions

If we took two 500 word articles and found that article A mentions 'Churchill' six times, and article B mentions 'Churchill' twelve times — should we view article A as half as relevant? Likely not.

BM25 solves this by modifying TF-IDF:

$$BM25(D, q) = \underbrace{\frac{f(q, D) * (k + 1)}{f(t, D) + k * (1 - b + b * \frac{D}{d_{avg}})}}_{TF} * \underbrace{log\left(\frac{N - N(q) + 0.5}{N(q) + 0.5} + 1\right)}_{IDF}$$

The BM25 formula.

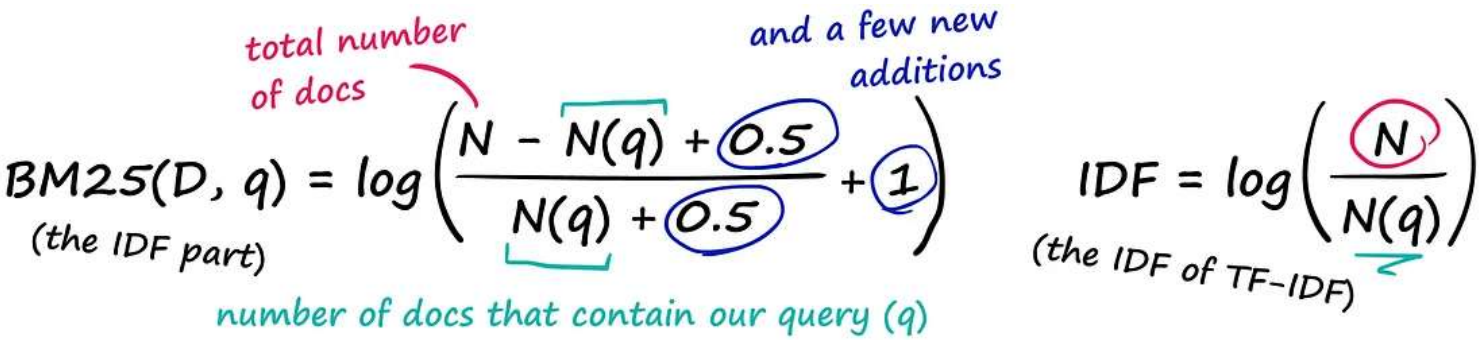
That's a pretty nasty-looking equation — but it's nothing more than our TF-IDF formula with a few new parameters! Let's compare the two TF components:

$$BM25(D, q) = \frac{f(q, D) * (k + 1)}{f(t, D) + k * (1 - b + b * \frac{D}{d_{avg}})} \qquad TF(D, q) = \frac{f(q, D)}{f(t, D)}$$

(the TF part) — same — new special parameters ($k = \sim 1.25$, $b = \sim 0.75$) — current document length — average document length — (the TF of TF-IDF)

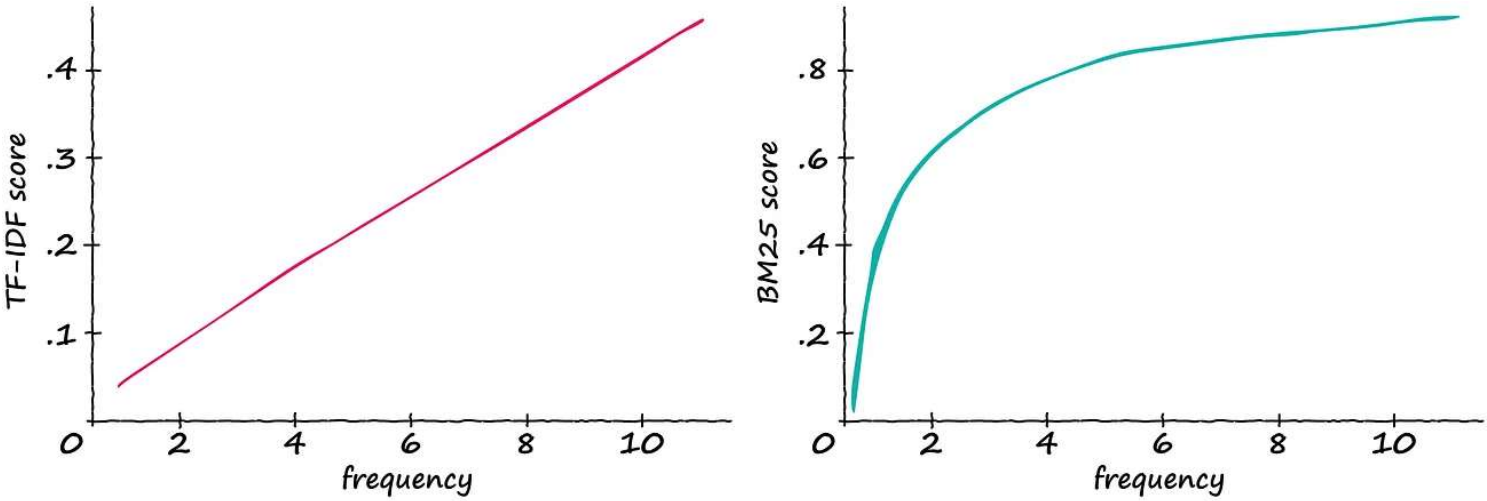
The TF part of BM25 (left) compared to the TF of TF-IDF (right).

And then we have the IDF part, which doesn't even introduce any new parameters — it just rearranges our old IDF from TF-IDF.




The IDF part of BM25 (left) compared to the IDF of TF-IDF (right).

Now, what is the result of this modification? If we take a sequence containing 12 tokens, and gradually feed it more and more 'matching' tokens — we produce the following scores:




Comparison of TF-IDF (left) and BM25 (right) algorithms using a sentence of 12 tokens, and an incremental number of relevant tokens (x-axis).

The TF-IDF score increases linearly with the number of relevant tokens. So, if the frequency doubles — so does the TF-IDF score.

BM25 dampens the score increase, we see an *x1.25* increase when doubling from two to four relevant tokens — and when we double again from four to

eight we see an increase of *x1.13*.

Sounds cool! But how do we implement it in Python? Again, we'll keep it nice and simple like the TF-IDF implementation.

We've used the default parameters for `k` and `b` — and our outputs look promising. The query `'purple'` only matches sentence `a`, and `'bananas'`

scores reasonable for both `b` and `c` — but slightly higher in `c` thanks to the smaller word count.

To build vectors from this, we do the exact same thing we did for TF-IDF.

Again, just as with our TF-IDF vectors, these are *sparse* vectors. We will not be able to encode semantic meaning — but focus on syntax instead. Let's take a

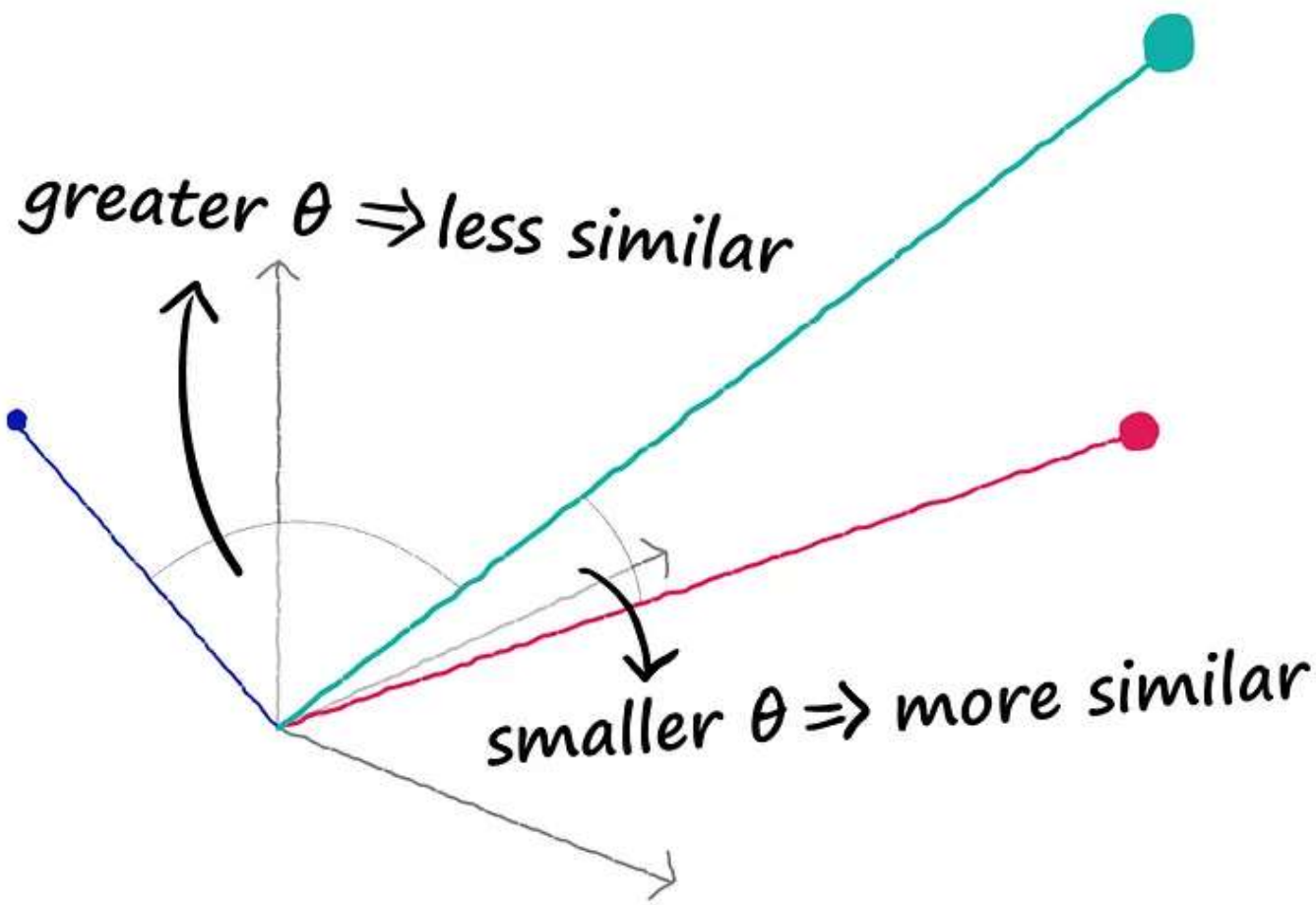
look at how we can begin considering semantics.

3. **BERT** — or Bidirectional Encoder Representations from Transformers — is a hugely popular transformer model used for *almost* everything in NLP.

Through 12 (or so) encoder layers, BERT encodes a huge amount of information into a set of *dense* vectors. Each dense vector typically contains 768 values — and we usually have 512 of these vectors for each sentence encoded by BERT.

These vectors contain what we can view as numerical representations of language. We can also extract those vectors — from different layers if wanted — but typically from the final layer.

Now, with two correctly encoded dense vectors, we can use a similarity metric like Cosine similarity to calculate their semantic similarity. Vectors that are more aligned are more semantically alike, and vise-versa.




A smaller angle between vectors (calculated with cosine similarity) means they are more aligned. For dense vectors, this correlates to greater semantic similarity.

But there's one problem, each sequence is represented by 512 vectors — not one vector.

So, this is where another — brilliant — adaption of BERT comes into play. Sentence-BERT allows us to create a single vector that represents our full sequence, otherwise known as a *sentence vector* [2].

We have two ways of implementing SBERT — the easy way using the `sentence-tranformers` library, or the slightly less easy way using `transformers` *and* PyTorch.

We'll cover both, starting with the `transformers` with PyTorch approach so that we can get an intuition for how these vectors are built.

If you've used the HF transformers library, the first few steps will look very familiar. We initialize our SBERT model and tokenizer, tokenize our text, and process our tokens through the model.

We've added a new sentence here, sentence **g** carries the same *semantic* meaning as **b** — without the same keywords. Due to the lack of shared words,

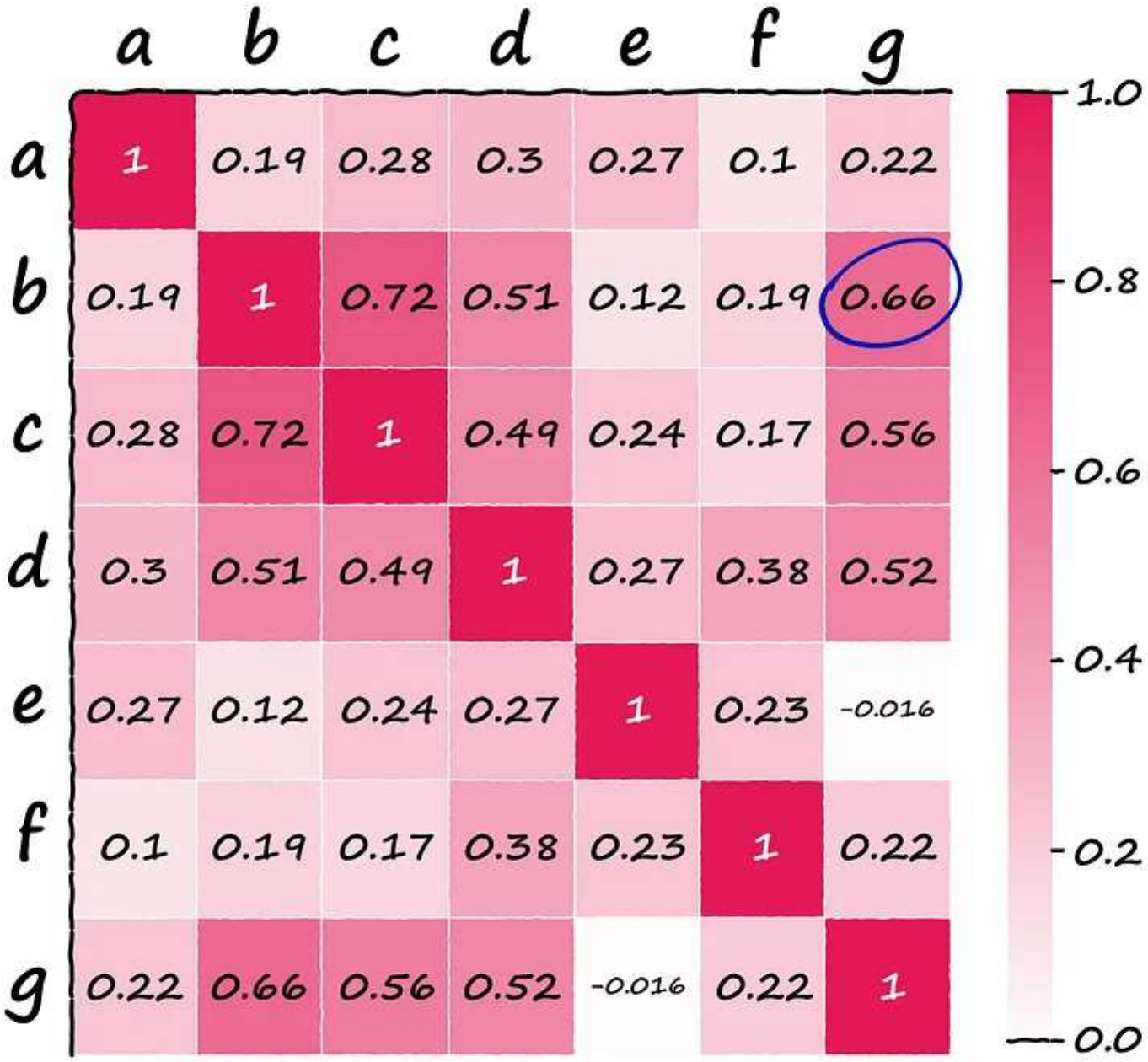
all of our previous methods would struggle to find similarity between these two sequences — remember this for later.

We have our vectors of length *768* — but these are **not** *sentence vectors* as we have a vector representation for each token in our sequence (128 here as we are using SBERT — for BERT-base this is 512). We need to perform a **mean pooling** operation to create the sentence vector.

The first thing we do is multiply each value in our `embeddings` tensor by its respective `attention_mask` value. The `attention_mask` contains **ones** where we have 'real tokens' (eg not padding tokens), and **zeros** elsewhere — this operation allows us to ignore non-real tokens.

And those are our sentence vectors, using those we can measure similarity by calculating the cosine similarity between each.

If we visualize our array, we can easily identify higher similarity sentences:




Heatmap showing cosine similarity between our SBERT sentence vectors — the score between sentences **b** and **g** is circled.

Now, think back to the earlier note about sentences **b** and **g** having essentially identical meaning whilst **not** sharing *any* of the same keywords.

We'd hope SBERT and its superior semantic representations of language to identify these two sentences as similar — and lo-and-behold the similarity between both is our second-highest score at **0.66** (circled above).

Now, **the alternative (easy) approach is to use sentence-transformers.** To get the exact same output as we produced above we write:

Which, of course, is much easier.

That's all for this walk through history with Jaccard, Levenshtein, and Bert!

We covered a total of **five** different techniques, starting with the straight-forward Jaccard similarity and Levenshtein distance. Before moving onto search with sparse vectors — TF-IDF and BM25, and finishing up with state-of-the-art dense vector representations with SBERT.

I hope you've enjoyed the article. Let me know if you have any questions or suggestions via Twitter or in the comments below. If you're interested in more content like this, I post on YouTube too.

Thanks for reading!

📚 Learn more about scalable search at Pinecone.io

## References

[1] Market Capitalization of Alphabet (GOOG), Companies Market Cap

[2] N. Reimers, I. Gurevych, Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks (2019), Proceedings of the 2019 Conference on Empirical Methods in 2019

Notebooks Repo

Colab for Jaccard| Levenshtein| TF-IDF | BM25 | SBERT

🤖 70% Discount on the NLP With Transformers Course

*All images are by the author except where stated otherwise*

Artificial Intelligence | Similarity Search | NLP | Machine Learning | Deep Dives

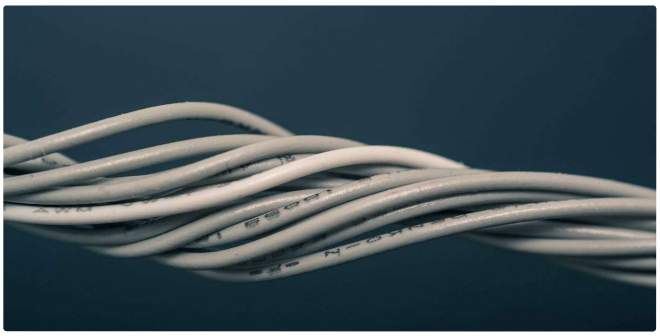

## Written by James Briggs

10.9K Followers · Writer for Towards Data Science

Freelance ML engineer learning and writing about everything. I post a lot on YT
https://www.youtube.com/c/jamesbriggs


Follow

### More from James Briggs and Towards Data Science



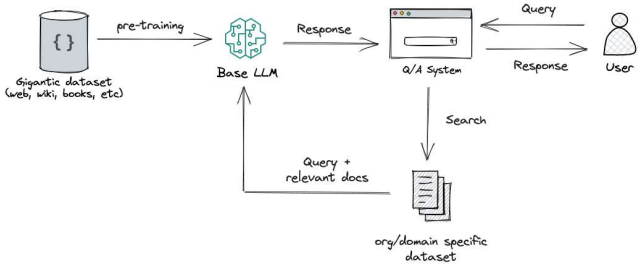
James Briggs in Towards Data Science

## The Right Way to Build an API with Python

All you need to know on API development in Flask

✦ · 7 min read · Sep 11, 2020

1.2K 13




Heiko Hotz in Towards Data Science

## RAG vs Finetuning—Which Is the Best Tool to Boost Your LLM...

The definitive guide for choosing the right method for your use case

✦ · 19 min read · Aug 25

2.1K 16

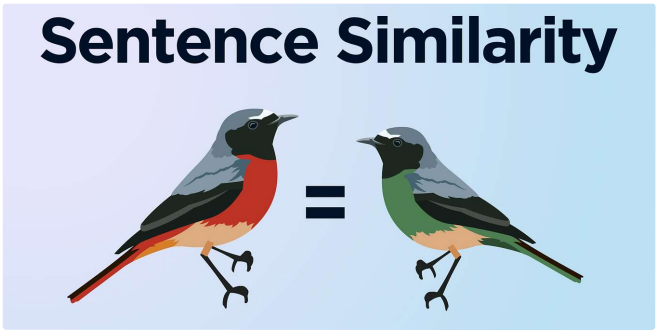
Giuseppe Scalamogna in Towards Data Science

## New ChatGPT Prompt Engineering Technique: Program Simulation

A potentially novel technique for turning a ChatGPT prompt into a mini-app.

9 min read · Sep 4

1.4K 14




James Briggs in Towards Data Science

## BERT For Measuring Text Similarity

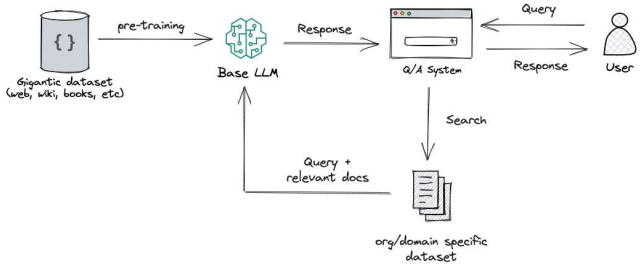
✦ · 5 min read · May 5, 2021

798 8

See all from James Briggs

See all from Towards Data Science

## Recommended from Medium



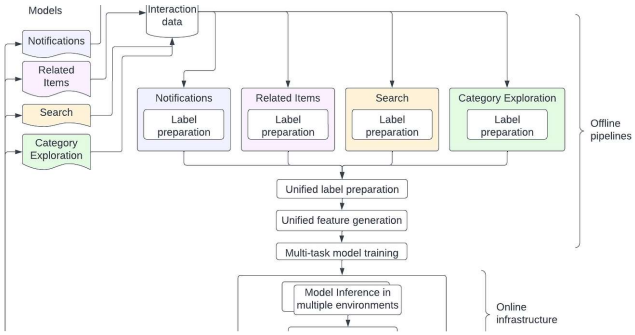

Heiko Hotz in Towards Data Science

### RAG vs Finetuning—Which Is the Best Tool to Boost Your LLM...

The definitive guide for choosing the right method for your use case

✦ · 19 min read · Aug 25

2.1K 16




Netflix Technology Blog

### Lessons Learnt From Consolidating ML Models in a Large Scale...

by Roger Menezes, Rahul Jha, Gary Yeh, and Sudarshan Lamkhede

7 min read · Aug 24

570 4

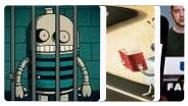
## Lists


**Natural Language Processing**

652 stories · 257 saves


**Predictive Modeling w/ Python**

20 stories · 428 saves



**AI Regulation**

6 stories · 136 saves

**ChatGPT prompts**

24 stories · 434 saves

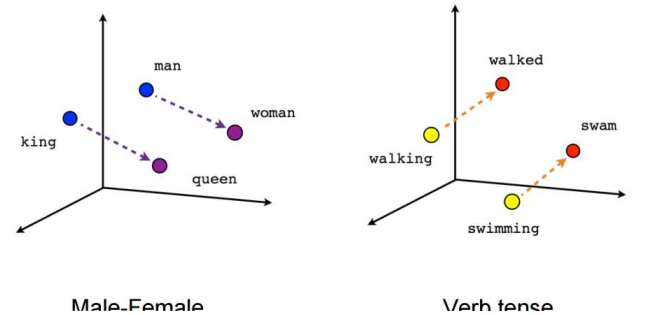
Jonathan Fulton in Jonathan's Musings

## AB Testing 101

What I wish I knew about AB testing when I started my career

17 min read · Aug 25

590 5




Male-Female Verb tense

Maninder Singh

## Accelerate Your Text Data Analysis with Custom BERT Word...

One thing is for sure the way humans interact with each other naturally is one of the most...

4 min read · Apr 24

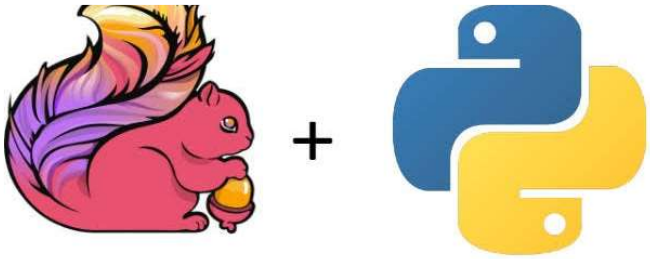
155

Alyx

## Semantic Search with FAISS

HuggingFace get_neareast_example and Cosine Similarity Search

9 min read · Jul 15



Daniel Santana

## Flink + Docker + Kafka

Apache Flink is a powerful stream processing framework that enables real-time data...

2 min read · Aug 7


See more recommendations