Using Softmax for Probability Transformation in Vector Search

Jaepil Jeong
Cognica, Inc.
Email: jaepil@cognica.io
Date: February 16, 2025

Introduction

Modern search systems often combine **vector search** (dense embedding retrieval) with traditional text search (BM25/TF-IDF) to improve result relevance. This is known as **hybrid search**, which leverages both semantic similarity and exact keyword matching. In practice, hybrid approaches have shown significant improvements in search quality across various domains and benchmarks [opensearch.org]. However, a key challenge in hybrid search is that dense and sparse retrieval methods produce **scores on different scales**. For example, a BM25 score might range from 0 to $15+$, while a cosine similarity from an embedding model ranges between -1 and 1. These raw scores are not directly comparable, making it hard to fairly merge results from the two systems.

The solution is to apply a **probability transformation** to the scores so that both vector and text results are represented in a common probabilistic scale. By converting scores into probabilities of relevance, we can combine and rank documents from different sources in a principled way. This approach is grounded in the **Probability Ranking Principle (PRP)** of information retrieval, which states that documents should be ordered by decreasing probability of relevance to the query [cseweb.ucsd.edu]. In other words, if we can estimate a probability that each document is relevant, we should rank by that.

In this article, we will focus on using the **softmax function** to transform similarity scores into probabilities for vector search. We will also compare softmax normalization to other methods like min-max scaling, sigmoid transformation, and Platt scaling, and show how these help in **hybrid search** (combining BM25/TF-IDF with vector search). The content is aimed at intermediate to advanced engineers, with a rigorous mathematical perspective and practical examples in Python.

Mathematical Theory

Similarity Measures in Vector Search

Before diving into softmax, let's clarify the common **similarity measures** used in vector search and how they produce scores:

- **Cosine Similarity:** Measures the cosine of the angle between two vectors. Given a query vector q and a document vector d, cosine similarity is defined as

$$\text{sim}_{\cos}(q, d) = \frac{q \cdot d}{\|q\| \, \|d\|}. \tag{1}$$

  This yields a value in $[-1, 1]$, where 1 means the vectors point in the same direction (very similar) and 0 means they are orthogonal (no similarity). In practice, vectors are often normalized to unit length so that cosine similarity is just the dot product of q and d. Higher cosine similarity means more relevance.

- **Dot Product (Inner Product):** The unnormalized similarity

$$q \cdot d = \sum_i q_i d_i. \tag{2}$$

  If vectors are not unit-normalized, the dot product can range more widely (including beyond 1). Many neural retrieval models use dot product as a score. A larger dot product indicates higher similarity/relevance. (If vectors are unit-length, dot product and cosine are equivalent.)

- **Euclidean Distance:** The distance between vectors,

$$\text{dist}(q, d) = \|q - d\|_2. \tag{3}$$

  Smaller distances mean the vectors are closer in the embedding space, thus *more* similar in content. Often vector databases use distance metrics (like Euclidean or its square, or Manhattan distance) for search. We can treat a **lower** distance as a higher relevance score by negating the distance.

Each of these metrics yields a **raw score** for a query-document pair. Cosine and dot product produce higher-is-better scores, whereas distance produces lower-is-better scores. When integrating with traditional IR scores (like BM25), which are also higher-is-better, we need to be careful to convert distances into a comparable form (usually by taking a negative distance or an inverse so that higher means more similar).

The Softmax Function for Score Normalization

The **softmax function** is a mathematical operation that converts a set of raw scores into a probability distribution. Given a list of scores $\{s_1, s_2, \ldots, s_n\}$ (these could be cosine similarities, dot products, BM25 scores, etc.), the softmax of these scores is defined as:

$$P(i) = \frac{\exp(s_i)}{\sum_{j=1}^{n} \exp(s_j)}, \quad \text{for } i = 1, \ldots, n. \tag{4}$$

This transforms the scores s_i into probabilities $P(i)$ that sum up to 1. After applying softmax, each $P(i)$ lies in the interval $(0, 1)$ and $\sum_{i=1}^{n} P(i) = 1$ [en.wikipedia.org]. Intuitively, $\exp(s_i)$ turns the score into a positive number, and dividing by the sum of all exponentials ensures normalization. The highest raw score will correspond to the highest probability (since a larger s_i yields a larger e^{s_i}), and differences in scores are magnified by the exponential. This means softmax **highlights the top-scoring items** more strongly than a linear scaling would.

Softmax is widely used in machine learning models (e.g., as the final layer in classifiers) to produce a probability distribution over classes [en.wikipedia.org]. In the context of search, we can think of each document in the result list as a "candidate class" for being the relevant result. Softmax then gives us a way to interpret the scores as a probability-like confidence. For example, if one document's score is much higher than all others for a query, softmax will assign it a probability close to 1 (and others near 0), indicating it's overwhelmingly likely to be the most relevant under that scoring scheme. Indeed, researchers have observed that when a query has a clear lexical match, the top BM25 result after softmax normalization can get a probability in the 0.8–1.0 range [arxiv.org] (dominating the distribution). In contrast, if the scores are closer together, softmax yields a more spread-out probability distribution among the results.

Why use softmax for normalization? One reason is that it puts scores from different scales onto a common probabilistic scale. BM25 scores and embedding similarities can then be compared apples-to-apples as probabilities. Another reason is that it aligns with the probability ranking principle: if we manage to approximate each document's relevance probability, ranking by softmax scores is theoretically optimal

under certain assumptions [cseweb.ucsd.edu]. Softmax is a **monotonic** transformation of the scores (it preserves the rank order of scores), so it won't change which document is ranked first, second, etc., for a given set. But it changes the **distribution** of scores, which is crucial when we later combine scores from different systems.

Importantly, softmax is typically applied **per query** on the set of candidate results for that query. You wouldn't apply softmax over the entire corpus (that would be infeasible); instead, you take the top k results from each subsystem (e.g., top 100 BM25 results) and normalize those scores via softmax. For example, Xiong et al. (2020) normalize the top-k BM25 scores using a softmax in their hybrid retrieval approach [arxiv.org]. This yields a probability distribution *over those k documents* for that query. Keep in mind that this distribution is relative to the considered candidates; if a truly relevant document isn't in the candidate set, these probabilities don't directly reflect absolute relevance, only relevance among the retrieved candidates.

Mathematically, one downside is that the softmax probabilities are **interdependent** – if you add more candidates or change a score, all probabilities shift since the denominator $\sum_j e^{s_j}$ changes. Despite this, softmax provides a convenient way to normalize scores from different scales and is easy to implement.

Temperature Scaling in Softmax

The softmax function can be adjusted by a **temperature parameter**, which controls the "peakiness" or spread of the output distribution. We introduce a parameter T (often denoted τ in literature) into the softmax as follows:

$$P(i;T) = \frac{\exp(s_i/T)}{\sum_{j=1}^{n} \exp(s_j/T)}. \tag{5}$$

- If $T = 1$, this is the standard softmax.
- If $T < 1$ (temperature less than 1, e.g. 0.5), it **sharpens** the distribution, making the highest score receive an even larger share of probability. In the limit as $T \to 0$, softmax approaches a hard-*argmax* (the highest-scoring item gets probability 1 and everything else 0).
- If $T > 1$, it **flattens** the distribution, reducing the differences between probabilities. As $T \to \infty$, $P(i;T)$ approaches a uniform distribution (all candidates get roughly equal probability).

In other words, the temperature acts as a smoothing factor: a low temperature emphasizes even small score differences, while a high temperature makes the model more "uncertain" or indifferent to score differences [aclanthology.org]. Temperature scaling is useful when you want to calibrate how much the score differences matter in the final probabilities. For example, you might find that your embedding model's raw scores produce an overly confident softmax distribution (one result gets $p \approx 0.99$). By using a higher T, you can flatten that out a bit, which might help when combining with another system. Conversely, if you want to trust a given score source more, you can use $T < 1$ to make its probability distribution more peaked (increasing the influence of its top result).

It's worth noting that temperature scaling is a common technique in machine learning for **probability calibration**. When combining multiple systems, you could treat T as a hyperparameter to tune for best performance. Setting T appropriately helps ensure that the probabilities from different systems are on comparable levels of confidence.

Comparison with Other Probability Transformations

Softmax is one way to normalize scores, but there are other approaches to transform raw scores into a 0–1 range or into probabilities. Each method has its theoretical underpinnings and practical trade-offs. Here we compare softmax to three other techniques: **min-max normalization**, the **sigmoid (logistic) transformation**, and **Platt scaling**.

Min-Max Normalization

Min-max normalization rescales the scores linearly into a fixed range, typically $[0, 1]$. Given a set of scores $\{s_i\}$ for a particular query, you can compute:

$$s_i' = \frac{s_i - s_{\min}}{s_{\max} - s_{\min}}, \tag{6}$$

where $s_{\max} = \max_j s_j$ and $s_{\min} = \min_j s_j$ among the candidates. The highest score becomes 1.0 and the lowest becomes 0.0, with others in between proportionally. This is a simple normalization that ensures all values are between 0 and 1.

Min-max normalization is easy to implement and preserves the ordering of scores (it's a monotonic linear transform). It's useful for making scores from different sources roughly comparable in scale. For example, OpenSearch's hybrid search allows a *min_max* normalization option for lexical and neural scores before combining [opensearch.org]. By scaling each subsystem's scores to $[0, 1]$, one can then combine them (e.g., via a weighted average) without one system's scores numerically dominating due to scale differences.

Trade-offs: Unlike softmax, min-max normalization does not produce a probability distribution that sums to 1; the values are just scaled scores. Thus, you can't directly interpret 0.8 as "80% probability of relevance" – it's just a relative score. Also, min-max is sensitive to outliers: if one document has an extremely high score, everything else will be squeezed towards 0. In practice, using a fixed top-k set per query mitigates this, since we're only normalizing within the top results. Another consideration is that min-max normalization is *local to the query's result set*. If a query returns generally low scores (e.g., no good match, so even the best score is mediocre), min-max will still stretch those so that the best $= 1.0$. This might overstate the confidence for a difficult query. Softmax has a similar issue (it will always assign 100% total probability across the considered docs), but it tends to penalize lower scores exponentially, whereas min-max will give a full 1.0 to the top result regardless of the gap.

Sigmoid (Logistic) Transformation

The **sigmoid function**, also known as the logistic function, is another way to map a real-valued score to a $[0, 1]$ range. The standard sigmoid is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{7}$$

If x is very large positive, $\sigma(x) \to 1$; if x is very large negative, $\sigma(x) \to 0$; and $\sigma(0) = 0.5$. To apply this to retrieval scores, one approach is to treat the raw score (say a dot product or BM25 value) as the input x. For example, one might say

$$P_{\text{sig}} = \frac{1}{1 + \exp(-s)}. \tag{8}$$

This would convert large positive scores to probabilities near 1, and large negative scores to probabilities near 0.

In practice, a raw retrieval score might not be centered around 0 or have a meaningful scale for the sigmoid. Often, an **affine transformation** is applied first: $\sigma(a \cdot s + b)$, where a and b are parameters that adjust the scale and offset. Without choosing a, b properly, a naive sigmoid might output probabilities that are all bunched around 0.5 (if s values are small) or saturate to $0/1$ (if s values are huge). This is where **calibration** comes in (like Platt scaling, discussed next).

Sigmoid transformation is essentially what you use in binary classification to get a probability of the positive class. If we had a predictor that gives a score for how relevant a document is, the sigmoid of that score could be interpreted as the probability that "document is relevant (class 1)". Unlike softmax, the sigmoid does **not** force the outputs of all documents to sum to 1. Each document's probability is evaluated independently. This means you can have multiple documents each with a reasonably high probability, which conceptually fits the scenario where several documents can all be relevant to the query (relevance is not a mutually exclusive single-choice outcome). In fact, if you treat each document's relevance as an independent Bernoulli event, a sigmoid score is an attempt to model

$$P(\text{relevant} \mid q, d) \tag{9}$$

for each doc individually, whereas softmax is modeling something like

$$P(d_i \text{ is most relevant} \mid q, \{d_j\}) \tag{10}$$

among a set of candidates.

The difference between these perspectives is important: **Softmax produces a *competition* among documents** (increasing one probability decreases the others, since they must sum to 1), whereas **sigmoid treats each document on its own**, allowing for the possibility that many documents are relevant. In an engineering context, if you want to simply rank results, both can work (they're monotonic transforms of the scores if calibrated well). But if you also want a meaningful relevance threshold ("only show docs with probability > 0.7"), a calibrated sigmoid is more suitable.

To illustrate, consider a scenario from a multi-label classification perspective: if we used softmax on a set of labels, we imply only one label can be correct. If multiple can be correct, we use independent sigmoid outputs. Similarly, in retrieval, using softmax strictly implies one "winner" per query, whereas using a sigmoid (or similar per-document probability) allows multiple winners. This is why, for example, one research paper notes that they used a sigmoid instead of softmax when they wanted to allow predicting multiple relevant terms/documents [arxiv.org].

Trade-offs: The sigmoid (without calibration) has an arbitrary scale. You might have to experiment with scaling the input (like dividing your scores by some constant) to get reasonable probability outputs. If not calibrated, a sigmoid's output might be systematically overly confident or under-confident. The benefit is that it's simple and can be applied globally or per query without needing the context of other documents (no need to sum over others). Also, combining probabilities from different sources multiplicatively is straightforward if each represents an independent probability (see hybrid section below). In contrast, combining softmax probabilities from different systems is less straightforward since each set sums to 1 separately.

Platt Scaling (Score Calibration)

Platt scaling is a specific technique for transforming scores into probabilities by learning a sigmoid function that maps the scores to observed probabilities. It was introduced by John Platt for converting SVM outputs into calibrated probabilities, but it's applicable generally to any scoring model. The idea is to take some held-out data (or cross-validation data) with ground truth relevances, and fit a logistic regression **with a single input feature (the raw score)**. In other words, find parameters A and B such that:

$$P_{\text{calibrated}} = \frac{1}{1 + \exp(A \cdot s + B)}, \tag{11}$$

best matches the actual relevances in a least-squares or log-loss sense. The parameters A, B are chosen to maximize the likelihood of the true labels given the scores, effectively training a tiny logistic regression where the model's score is the only input. Platt's original method also includes some tricks to avoid overfitting (like adding prior counts to the data) [stats.stackexchange.com].

In summary, Platt scaling **uses logistic regression to map an uncalibrated score to a probability** [stats.stackexchange.com]. You supply it with examples of scores for known relevant and non-relevant documents; it finds the best sigmoid function to separate those. After Platt scaling, the output can be interpreted as an estimate of

$$P(\text{relevant} \mid \text{score}). \tag{12}$$

For instance, if your BM25 score is 7.3 for a document and after Platt scaling that corresponds to 0.65, you can interpret that as roughly a 65% chance of relevance (assuming your training data was representative).

Trade-offs: Platt scaling typically yields **excellent calibration** of probabilities if the model's scores have some monotonic relationship with relevance. It will make, say, a BM25 score of 10 mean the same level of confidence as a vector similarity of 0.8, provided both were calibrated on relevance judgments. This makes combining scores trivial: you could just compare calibrated probabilities or average them. The downside is that it requires training data (relevance labels for queries). If your search system doesn't have training queries and labeled relevant documents, you can't directly do Platt scaling. Also, Platt scaling is a **global** method – it doesn't change per query. It assumes the relationship between score and probability is consistent across queries.

In practice, Platt scaling is widely used in machine learning competitions to calibrate classifier probabilities and improve log-loss without changing classification accuracy [analyticsvidhya.com]. In IR, if you have click data or judgments, you could use it to calibrate, say, a neural retriever's dot product to probability of click vs. skip. Another caution is that Platt scaling (or any calibration) will **not** significantly alter the rank order if the underlying model is good – it mostly affects the confidence estimates. As an example, after Platt scaling, you might not see big changes in rank metrics like NDCG or precision (since those depend on ordering), but you would see improvements in a metric like log loss or in the interpretability of the scores.

To summarize the comparisons:

- **Softmax:** Produces a probability distribution *across candidates* (relative relevance). Good for ensuring a common scale and $\sum = 1$ normalization; highlights the top result strongly. Needs to be done per query on a set of candidates. Doesn't provide an independent probability for each document being relevant (depends on others in the set).
- **Min-Max:** Simple linear scaling to $[0, 1]$. Preserves order, no probabilistic meaning, but quick to compute. Can be used as a preprocessing step before combining scores (e.g., to limit them to a range).

- **Sigmoid:** (Logistic) Provides independent probabilities for each item. Without calibration, requires guessing a scale; with calibration (Platt), can yield accurate probabilities. Allows multiple high-probability docs.
- **Platt Scaling:** A learned sigmoid fit to data. Best for calibration if you have data; yields meaningful absolute probabilities of relevance from scores. Essentially a tuned version of the sigmoid per model.

In practice, you choose the method based on available data and requirements. If you just need to merge scores from two systems and have no training data, softmax or min-max are common choices to normalize scales [opensearch.org]. If you have lots of click or relevance data, calibrating each system's scores via Platt scaling and then combining might give the most theoretically sound results (since each score is an estimate of $P(\text{relevance})$). It's also possible to combine approaches: for example, first apply a sigmoid to each score with some hand-tuned parameters to roughly calibrate it, then apply a softmax to a pool of candidates from both systems.

Hybrid Search: Combining Text and Vector Search

Hybrid search refers to the integration of **sparse retrieval** (e.g., BM25, TF-IDF) and **dense retrieval** (embeddings) to leverage their complementary strengths. Sparse methods excel at precise keyword matching and handling rare terms, while dense methods excel at capturing semantic similarity and contextual meaning [weaviate.io]. A hybrid system can return results that a purely lexical search would miss due to vocabulary mismatch, while still giving high priority to exact matches that a purely semantic search might overlook.

The main challenge in hybrid search is **score integration**: BM25 scores and vector similarity scores are not directly comparable. BM25 scores are based on term frequencies and document length normalization, whereas embedding similarities come from vector dot products or distances in some high-dimensional space. There is no inherent way to say, for example, that a BM25 score of 12 is better or worse than a cosine similarity of 0.8. Without normalization, if you simply took a weighted sum

$$\text{Score}_{\text{combined}} = w \times \text{BM25} + (1 - w) \times \text{Score}_{\text{embedding}}, \tag{13}$$

the choice of weight w would be arbitrary and not generalizable (since the scales differ by orders of magnitude and distribution).

By converting each system's output into **probabilistic scores**, we get a common ground for comparison. Softmax-based normalization is particularly handy here. Here's how a hybrid search pipeline might work using softmax probabilities:

1. **Retrieve candidates from each system:** For a given query, do a BM25 search (sparse) and a vector search (dense). Suppose you take the top k results from each. You now have two lists of documents with their raw scores.
2. **Normalize scores to probabilities:** Apply a softmax to the BM25 scores from those top k (separately from the rest of the corpus) to get a probability distribution

   $$P_{\text{BM25}}(d_i \mid q) \tag{14}$$

   over the BM25 candidates [arxiv.org]. Do the same for the embedding scores to get

   $$P_{\text{vec}}(d_j \mid q) \tag{15}$$

over the vector candidates. At this point, you have two probability distributions, each summing to 1 over their respective candidate sets. (Alternatively, you could include all candidates from both lists in one big set and apply softmax to the union, but doing it separately is common and allows weighting them differently.)

3. **Fuse the distributions:** Now you combine these probabilities from the two sources. There are a few strategies for this **score fusion** using the probabilistic outputs:
   - **Weighted Linear Combination (Mixture):** This is one of the simplest and most effective methods. You decide on a weight (let's call it α) between 0 and 1 for the dense vs. sparse contributions. Then for each document in the combined candidate set (the union of both lists), you compute:

     $$P_{\text{combined}}(d \mid q) = \alpha \cdot P_{\text{vec}}(d \mid q) + (1 - \alpha) \cdot P_{\text{BM25}}(d \mid q). \tag{16}$$

     If a document appears in only one of the two lists, the missing probability is treated as 0 in that list. The combined probabilities may then be renormalized to sum to 1 (though if $\alpha + (1 - \alpha) = 1$ and both distributions summed to 1 separately, the sum of combined probabilities will actually still be 1 — it's effectively a **mixture of two distributions**). This approach is straightforward and interpretable: α is a knob that tunes the relative influence of semantic vs. lexical search. An α of 0.5 gives equal weight. A higher α favors vector similarity more, while a lower α leans on BM25 more. OpenSearch, for example, uses a weighted sum (arithmetic mean) of normalized scores for hybrid search, with user-configurable weights [opensearch.org].
   - **Multiplicative Fusion (Product):** Another strategy is to assume the two systems provide independent evidence of relevance and multiply the probabilities:

     $$P_{\text{combined}}(d \mid q) \propto P_{\text{vec}}(d \mid q) \times P_{\text{BM25}}(d \mid q). \tag{17}$$

     After multiplying for each document (treat missing as 0 or a very small ϵ), you would renormalize so that probabilities sum to 1. This approach is akin to a Bayesian update or **naïve Bayes combination** where the embedding and lexical scores are independent signals. The effect of multiplication is that a document needs to be considered relevant by *both* models to get a high combined score. If either component gives a low probability (e.g., BM25 thinks the document isn't relevant, so P_{BM25} is near 0), then the product will be near 0. This can improve precision (both signals have to agree), but might hurt recall (a document highly ranked by one method alone will be suppressed if the other method missed it). In practice, pure multiplicative fusion can be too strict unless both systems are fairly complete. Sometimes a slight variant is used:

     $$P_{\text{combined}} = P_{\text{BM25}}^{\beta} \cdot P_{\text{vec}}^{(1-\beta)} \tag{18}$$

     (geometric mean of probabilities) which is similar to multiplication but with a power to weight one more than the other; this needs renormalization too. Interestingly, taking a geometric mean corresponds to a form of score combination as well (OpenSearch offers a geometric mean fusion option) [opensearch.org].
   - **Other Fusion Methods:** There are more heuristic methods like **Reciprocal Rank Fusion (RRF)** which do not directly use probabilities but rather rank positions. RRF, for instance, assigns a score like

$$\frac{1}{k + \text{rank}} \tag{19}$$

for each result from each list and sums them [weaviate.io]. While RRF is very robust and simple, it doesn't give a probabilistic interpretation. Another approach could be a learned ranker that takes the scores from both systems as features and learns how to combine them (which could implicitly learn something akin to a weighted combination or even a non-linear combination). For the scope of this post, we'll stick to probabilistic fusion strategies, as they are intuitive and align with the idea of **probability of relevance**.

After fusion, you have a single list of documents with combined probability scores. The final step is simply to **rank documents by** $P_{\text{combined}}(d \mid q)$ **in descending order** and take the top N as the hybrid search results.

A big advantage of converting to probabilities and then combining is **seamless integration**: each system contributes according to how confident it is. For example, if the BM25 subsystem finds an exact match (making one document's BM25-softmax probability 0.9), and the embedding subsystem finds only loosely related documents (its probabilities are spread out, say max 0.4), a sensible α will ensure the exact match is ranked first because 0.9 from BM25 outweighs the others. Conversely, if BM25 finds nothing (all low scores) but the embedding finds a very relevant doc, the probabilities will reflect that, and the embedding result can take the top spot. Softmax here is crucial because it naturally **scales the scores**: a BM25 score of 15 vs. 10 might turn into probabilities 0.85 vs. 0.15 (very confident distinction), whereas an embedding score of 0.85 vs. 0.8 might turn into probabilities like 0.52 vs. 0.48 (nearly tied). These normalized values give a meaningful way to compare the confidence of the two systems. In fact, research has noted patterns like: when BM25 scores are high (strong lexical overlap), the softmax-normalized BM25 probability for the top doc is very high (approximately 0.8–1), but when lexical scores are low, that probability is much lower [arxiv.org]. The hybrid system can use this information to lean on the vector search in the latter case.

Score fusion example: Suppose BM25 returns Document A with score 12 and Document B with score 8 for a query, while the vector search returns Document B with similarity 0.90 and Document C with 0.85. These raw scores aren't directly comparable. If we softmax-normalize, BM25 might give $P(A) = 0.88$, $P(B) = 0.12$; the vector side might give $P(B) = 0.53$, $P(C) = 0.47$. Now if we take a simple average ($\alpha = 0.5$), Document A gets combined probability 0.44 (0.88×0.5 from BM25 + 0 from vector since A wasn't in vector list), Document B gets 0.325 ($0.06 + 0.265$), and Document C gets 0.235 ($0 + 0.235$). In this case, Document A would still rank #1 (because BM25 was very confident about A), but Document B comes next, benefiting from being present in both lists, and Document C is third. If we had instead multiplied probabilities, Document A would get 0 (since it had 0 from vector), Document B would get $0.12 \times 0.53 = 0.0636$, and Document C would get 0 (since 0 from BM25), and after renormalizing, Document B would be top (A and C would essentially be dropped). This shows how different fusion strategies can dramatically affect results – weighted sum is more forgiving, while product is stricter.

In practice, **tuning the fusion parameters** (like α or the decision to multiply vs. add) is important. Many systems simply try a few weights and pick one that maximizes validation set performance. Many systems, such as OpenSearch's hybrid search, even provide an automated optimization to find the best global weights and normalization technique for a given dataset [opensearch.org] [opensearch.org]. The good news is that by using probability-based scores, these parameters tend to generalize better than if we were combining raw scores, because the ranges and meanings are aligned.

To summarize, softmax-based probability transformation enables hybrid search by providing a common probabilistic scale for heterogeneous scoring systems. Once every candidate has a probability of relevance, combining results from text and vector search becomes as simple as mixing or multiplying probabilities — a conceptually clean approach. This allows development of hybrid systems that are **versatile** (you can plug in additional signals like click-through rates, pagerank, etc., also as probabilities) and **robust** (each subsystem contributes when it has strength). As hybrid search becomes more prevalent, these normalization techniques form the foundation of how we merge the old and new paradigms of retrieval.

Implementation in Python

Let's put these ideas into practice with some Python code. We'll demonstrate how to apply the softmax transformation to different similarity metrics and then how to perform a simple hybrid score fusion. We will use NumPy for numerical computations. (In a real system, these calculations might be done in the search engine or database, but here we'll illustrate the concepts in code.)

Softmax normalization for various metrics:

```python
import numpy as np

def softmax(scores, temperature=1.0):
    """Compute softmax probabilities with an optional temperature scaling."""
    scores = np.array(scores, dtype=float)
    # Apply temperature scaling
    scaled_scores = scores / temperature
    # Subtract max for numerical stability (optional but good practice)
    scaled_scores -= np.max(scaled_scores)
    exp_scores = np.exp(scaled_scores)
    return exp_scores / np.sum(exp_scores)

# Example raw scores from different similarity metrics
cosine_similarities = [0.8, 0.5, 0.3]      # higher is better
dot_product_scores = [12.5, 10.2, 8.7]     # higher is better (e.g., unnormalized embedding scores)
euclidean_distances = [5.2, 7.1, 9.3]      # lower is better

# Softmax for cosine similarities
cosine_probs = softmax(cosine_similarities)
# Softmax for dot product scores
dot_probs = softmax(dot_product_scores)
# Softmax for Euclidean distances (use negative distance as similarity)
euclid_similarities = -np.array(euclidean_distances)
euclid_probs = softmax(euclid_similarities)

print(f"Cosine similarity probabilities: {cosine_probs}")
print(f"Dot product probabilities: {dot_probs}")
print(f"Euclidean distance probabilities (after inversion): {euclid_probs}")

# Demonstrate the effect of temperature scaling on a score list
scores = [2.0, 1.0, 0.1]  # example scores
for T in [0.5, 1.0, 2.0]:
    print(f"Softmax with T={T}: {softmax(scores, temperature=T)}")
```

Hybrid score fusion example:

```python
import numpy as np

def softmax(scores, temperature=1.0):
    """Compute softmax probabilities with an optional temperature scaling."""
    scores = np.array(scores, dtype=float)
    # Apply temperature scaling
    scaled_scores = scores / temperature
    # Subtract max for numerical stability (optional but good practice)
    scaled_scores -= np.max(scaled_scores)
    exp_scores = np.exp(scaled_scores)
    return exp_scores / np.sum(exp_scores)

# Example BM25 and vector search scores for a query
bm25_scores = {
    "DocA": 7.5,
    "DocB": 5.2,
    "DocC": 2.0
}
vector_scores = {
    "DocB": 0.88,
    "DocD": 0.77,
    "DocA": 0.65
}

# 1. Softmax normalize the scores within each result set
bm25_docs = list(bm25_scores.keys())
bm25_vals = list(bm25_scores.values())
bm25_probs = softmax(bm25_vals)
bm25_prob_dict = {doc: p for doc, p in zip(bm25_docs, bm25_probs)}

vector_docs = list(vector_scores.keys())
vector_vals = list(vector_scores.values())
vector_probs = softmax(vector_vals)
vector_prob_dict = {doc: p for doc, p in zip(vector_docs, vector_probs)}

print(f"BM25 probabilities: {bm25_prob_dict}")
print(f"Vector probabilities: {vector_prob_dict}")

# 2. Combine the distributions with a weighted sum
alpha = 0.5  # weight for vector model; (1-alpha) is weight for BM25
combined_scores = {}
# Union of all docs from both lists
all_docs = set(bm25_prob_dict.keys()) | set(vector_prob_dict.keys())
for doc in all_docs:
    p_vec = vector_prob_dict.get(doc, 0.0)
    p_bm25 = bm25_prob_dict.get(doc, 0.0)
    combined_scores[doc] = alpha * p_vec + (1 - alpha) * p_bm25

# 3. (Optional) Renormalize combined scores to sum to 1
```

```
total = sum(combined_scores.values())
combined_probs = {doc: score / total for doc, score in combined_scores.items()}

# 4. Sort documents by combined probability
ranked_results = sorted(combined_probs.items(), key=lambda x: x[1], reverse=True)
print(f"Combined probabilities: {combined_probs}")
print(f"Ranked results: {ranked_results}")
```

Let's break down what's happening here:

- We have a dictionary of BM25 scores for documents "DocA", "DocB", "DocC", and a dictionary of vector similarity scores for "DocB", "DocD", "DocA". Notice that "DocA" and "DocB" are returned by both systems (overlap), while "DocC" is only in BM25 results and "DocD" only in vector results.
- We apply softmax to each set of scores. $\mathrm{bm25\_prob\_dict}$ might end up looking something like `{"DocA": 0.90, "DocB": 0.09, "DocC": 0.01}` (if DocA's score is far above the others, as it is). $\mathrm{vector\_prob\_dict}$ might look like `{"DocB": 0.37, "DocD": 0.33, "DocA": 0.30}` (since the vector scores are relatively close). These are the per-system probability distributions P_{BM25} and P_{vec}. We print them out to see how the raw scores were normalized.
- We then choose a weight $\alpha = 0.5$ (equal blend) and compute

$$\mathrm{combined\_scores}[\mathrm{doc}] = 0.5 \times P_{\text{vec}}(\mathrm{doc}) + 0.5 \times P_{\text{BM25}}(\mathrm{doc}) \tag{20}$$

  for each document. If a document is missing in one of the distributions, we use 0 for that part. After this, $\mathrm{combined\_scores}$ might be:
  - DocA: $0.5 \times 0.30 + 0.5 \times 0.90 = 0.60$
  - DocB: $0.5 \times 0.37 + 0.5 \times 0.09 = 0.23$
  - DocC: $0.5 \times 0 + 0.5 \times 0.01 = 0.005$
  - DocD: $0.5 \times 0.33 + 0.5 \times 0 = 0.165$
    (These numbers will vary depending on the actual softmax outputs, but this is the general idea.)
- We optionally renormalize the combined scores to make them sum to 1 (for interpretation as probabilities). In this case, they already sum to 1 because of how a mixture of distributions works, but doing it explicitly is fine. We then sort the documents by the combined probability.

The final $\mathrm{ranked\_results}$ list, in this example, would be:

```
[('DocA', 0.60), ('DocB', 0.23), ('DocD', 0.165), ('DocC', 0.005)]
```

This indicates the hybrid system ranking: Document A is ranked #1 with a probability of 0.60, Document B #2 with 0.23, Document D #3 with 0.165, and Document C last with essentially 0.5% probability. This makes sense given our input: Document A was very strongly favored by BM25 (and had some vector support), so it wins. Document B had moderate scores in both, giving it a decent combined score. Document D was only in the vector results, getting a moderate probability, and Document C was only in BM25 but with a very low BM25 score, so it falls off.

You can experiment with changing α. For example, if we set α higher (say 0.8 for vector and 0.2 for BM25), Document D's contribution would increase and Document A's would decrease, possibly altering the order if extreme. You could also try the multiplicative approach: e.g.,

$$\text{combined\_scores}[\text{doc}] = P_{\text{vec}}(\text{doc}) \times P_{\text{BM25}}(\text{doc}) \tag{21}$$

and see how the ranking changes (Document A would actually get 0 in that case due to missing from the vector list, as discussed).

This simple demo shows how to implement softmax normalization and fusion. In a real-world scenario, you might integrate this logic into the search engine or perform it as a post-processing step of the query. Libraries and search frameworks may also have built-in support for hybrid scoring (for example, Elastic/OpenSearch's rank fusion, or vector databases that support hybrid queries with weighting parameters).

Conclusion

In this post, we explored the use of softmax for probability transformation in vector search, especially as a tool for hybrid search systems that combine lexical and semantic signals. We started by reviewing similarity measures (cosine, dot product, Euclidean) and highlighted the need to normalize their outputs when fusing different retrieval methods. The softmax function provides a mathematically principled way to turn arbitrary scores into a probability distribution, allowing us to interpret scores as confidence levels. We saw how introducing a temperature parameter adds flexibility in tuning the output distribution's entropy, which can be useful for calibration.

We compared softmax normalization to other approaches:

- **Min-max normalization** linearly scales scores to $[0, 1]$ without giving them probabilistic meaning, but it's simple and used in systems like OpenSearch for hybrid score blending.
- **Sigmoid (logistic) transformation** provides independent probabilities for each result, suitable when multiple results can be relevant. It doesn't enforce a fixed sum, which aligns with the reality of search (many documents can be relevant). However, it typically requires calibration or good parameter choices to yield meaningful probabilities.
- **Platt scaling** takes calibration to the next level by learning a mapping from scores to probabilities using logistic regression. This can yield accurate probability estimates of relevance if you have training data, enabling theoretically optimal ranking by probability. The trade-off is the need for labeled data and the assumption that the score-to-relevance mapping is consistent.

When it comes to hybrid search, probability transformations are the bridge that connects heterogeneous systems. By converting BM25 and vector scores into probabilities, we achieve a common scale on which to combine them. This makes it easy to apply score fusion strategies like weighted sums (a mixture of distributions) or products (assuming independent evidence) to get a single ranked list. We discussed how weighted combination is a flexible and often effective method to integrate results, and how tuning the weights or using different fusion formulas can balance precision and recall in hybrid retrieval.

In our Python examples, we demonstrated how to implement softmax normalization and combine two sets of results. The example illustrated in simple terms how a strong signal from one system can dominate or how an overlapping document benefits from both sources. In practice, you would validate such approaches on real queries – for instance, measure if an $\alpha = 0.7$ yields better mean average precision than $\alpha = 0.5$, or if a certain temperature on the softmax improves the NDCG of the merged results.

Key takeaways and best practices:

- Always **normalize scores** when combining different retrieval methods. Softmax is a good default for normalization because it provides a clear probabilistic interpretation and accentuates high-confidence results. At the very least, do something like min-max or z-score normalization to avoid apples-to-oranges comparisons of scores.
- Use temperature scaling or weights to calibrate the influence of each system. Treat these as hyperparameters. A dense vector model and a BM25 model might have different "confidence" levels; adjusting temperature or combination weight can compensate for that.
- Understand the interpretation of your scores. If you need a strict probability of relevance for each document, consider techniques like Platt scaling to calibrate scores against ground truth. If you only care about ranking, relative normalization (softmax) might suffice, but be mindful of cases where relative probabilities can be misleading (e.g., if all candidates are actually poor, softmax still assigns a total of 1.0 among them).
- Test different fusion strategies. Start with a simple weighted sum of probabilities (which is essentially what many production systems do, sometimes under the hood). But also try multiplicative fusion or more advanced ensembling if you have reason to believe one method's positives should only count if confirmed by the other.
- Maintain a fallback for edge cases. For example, if one of the systems returns no results (or the softmax distribution is very flat), ensure your combination logic can handle it (perhaps default to the other system entirely in such cases, or use a prior).

In conclusion, using softmax to transform vector search scores to probabilities is a powerful technique that not only helps in hybrid search combinations but also provides a deeper understanding of the model's confidence. It brings us closer to the ideal of the probability ranking principle by enabling scoring functions to speak a common language of "likelihood of relevance." As you implement or tune your own hybrid search system, apply these transformations and strategies, and you'll likely see more robust and interpretable results. Happy searching!