Building LLM Applications: Search & Retrieval (Part 5)

Vipra Singh · Follow

26 min read · Jan 27, 2024

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generation (RAG) Application.

Posts in this Series

1. *Introduction*
2. *Data Preparation*
3. *Sentence Transformers*
4. *Vector Database*
5. *Search & Retrieval (This Post)*
6. *LLM*
7. *Open-Source RAG*

8. *Evaluation*
9. *Serving LLMs*
10. *Advanced RAG*

. . .

Table of Contents

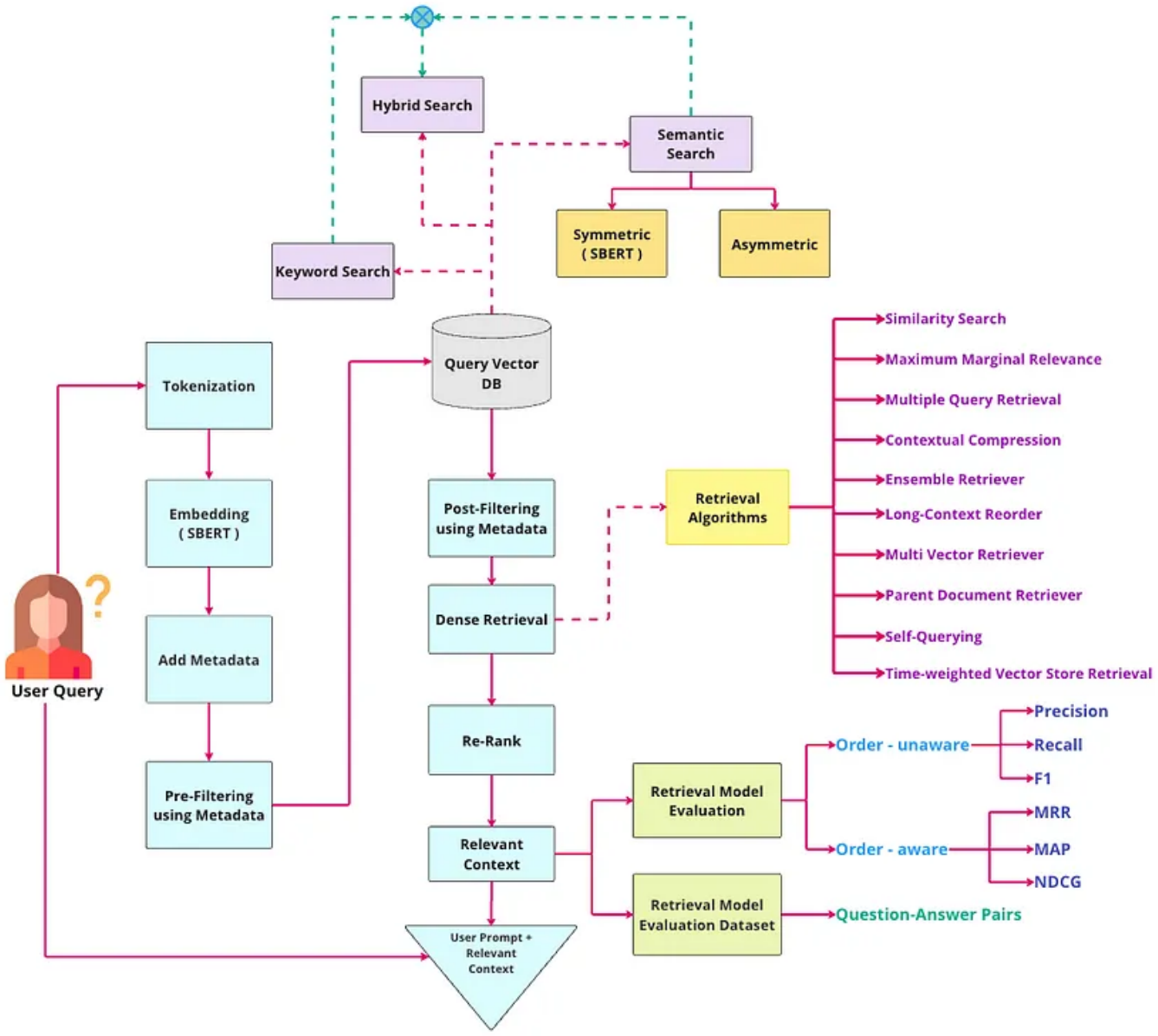


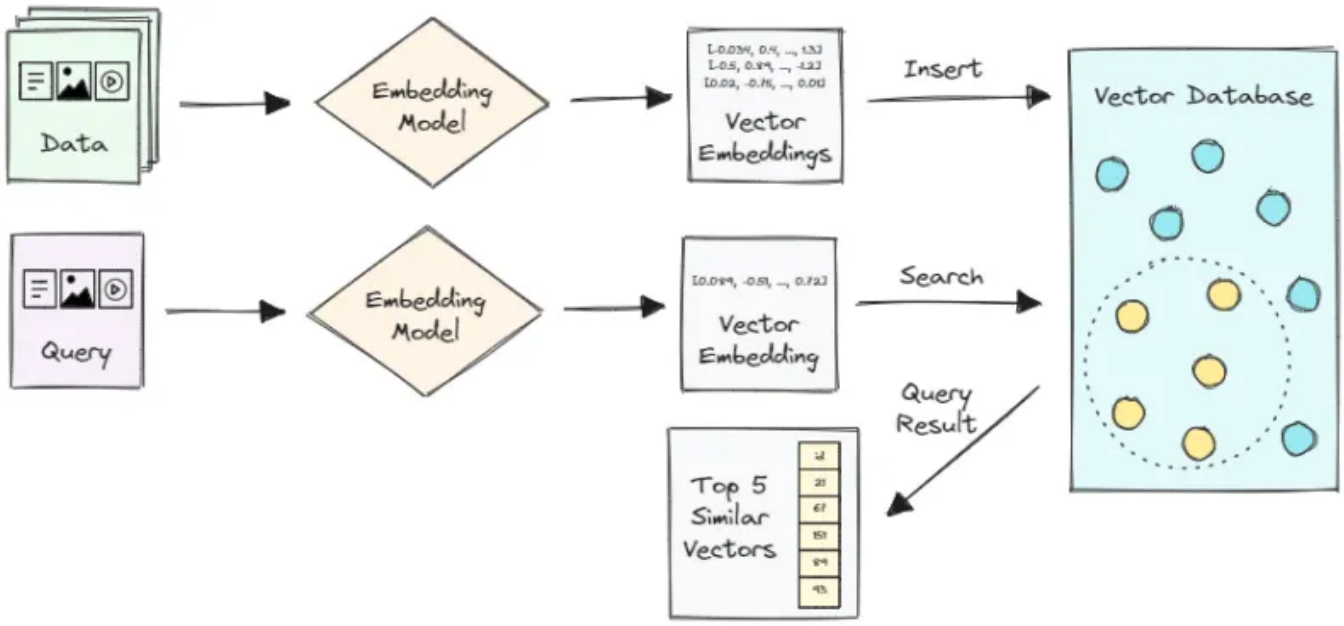
Image by Author

Greetings!

Let’s kick off our exploration into the search for pertinent data within the

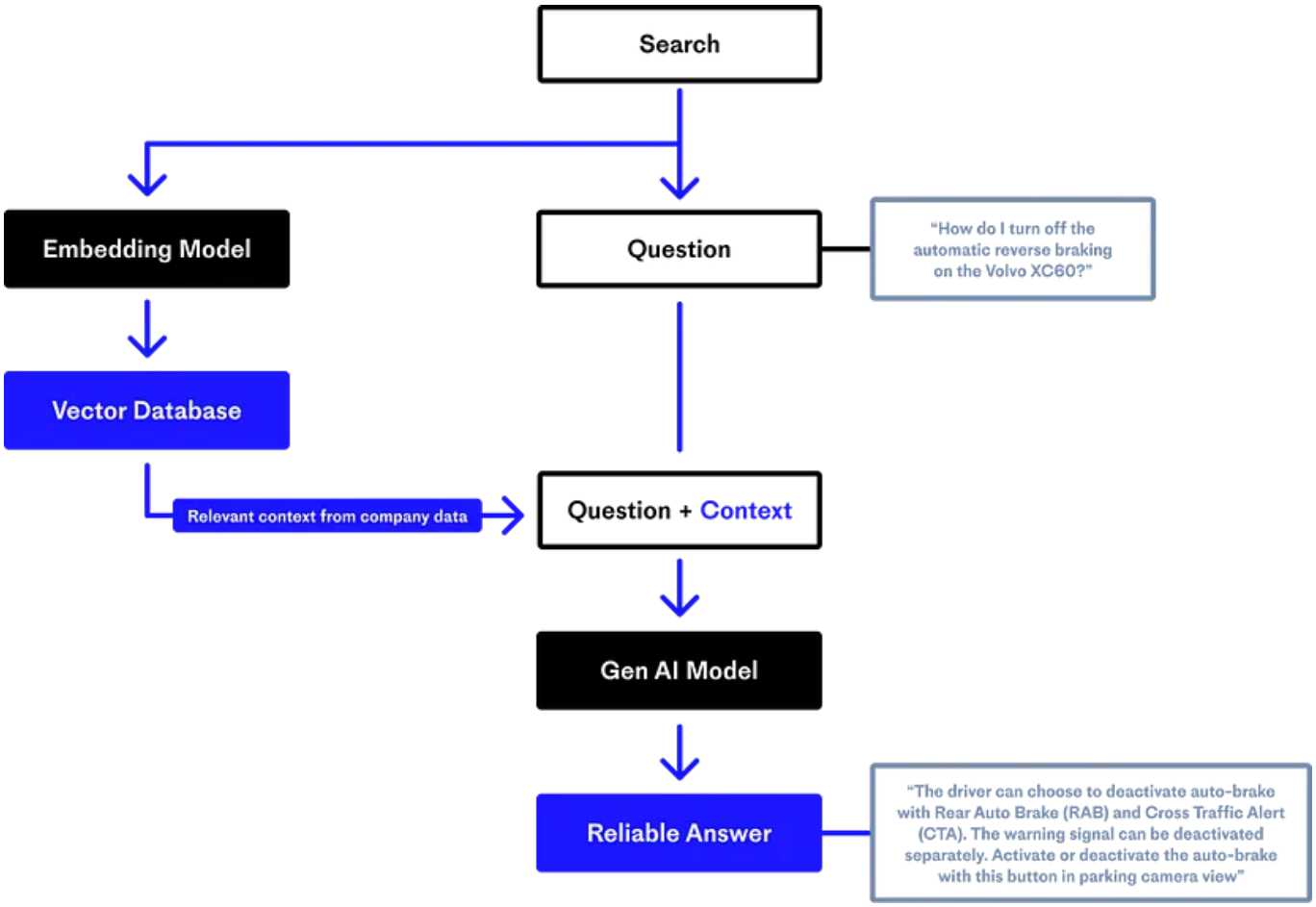
RAG Application.

When a user inputs a query, the process involves tokenizing the user query and performing embedding using the identical model utilized for embedding raw data. Subsequently, relevant chunks are extracted from the knowledge base, guided by their similarity to the user's query.



Vector DB

This blog will take a comprehensive look into the intricacies of the search process. The below figure shows where exactly "Search" fits in the entire RAG Pipeline.



Retrieval Augmented Generation reduces the likelihood of hallucinations by providing domain-specific information through an LLM's context window.

1. Introduction

Let's consider the common scenario of developing a customer support chatbot using an LLM. Usually, teams possess a wealth of product documentation, which includes a vast amount of unstructured data detailing their product, frequently asked questions, and use cases.

This data is broken down into pieces through a process called "chunking." After the data is broken down, each chunk is assigned a unique identifier and embedded into a high-dimensional space within a vector database. This process leverages advanced natural language processing techniques to understand the context and semantic meaning of each chunk.

When a customer's question comes in, the LLM uses a retrieval algorithm to quickly identify and fetch the most relevant chunks from the vector database. This retrieval is based on the semantic similarity between the query and the chunks, not just keyword matching.

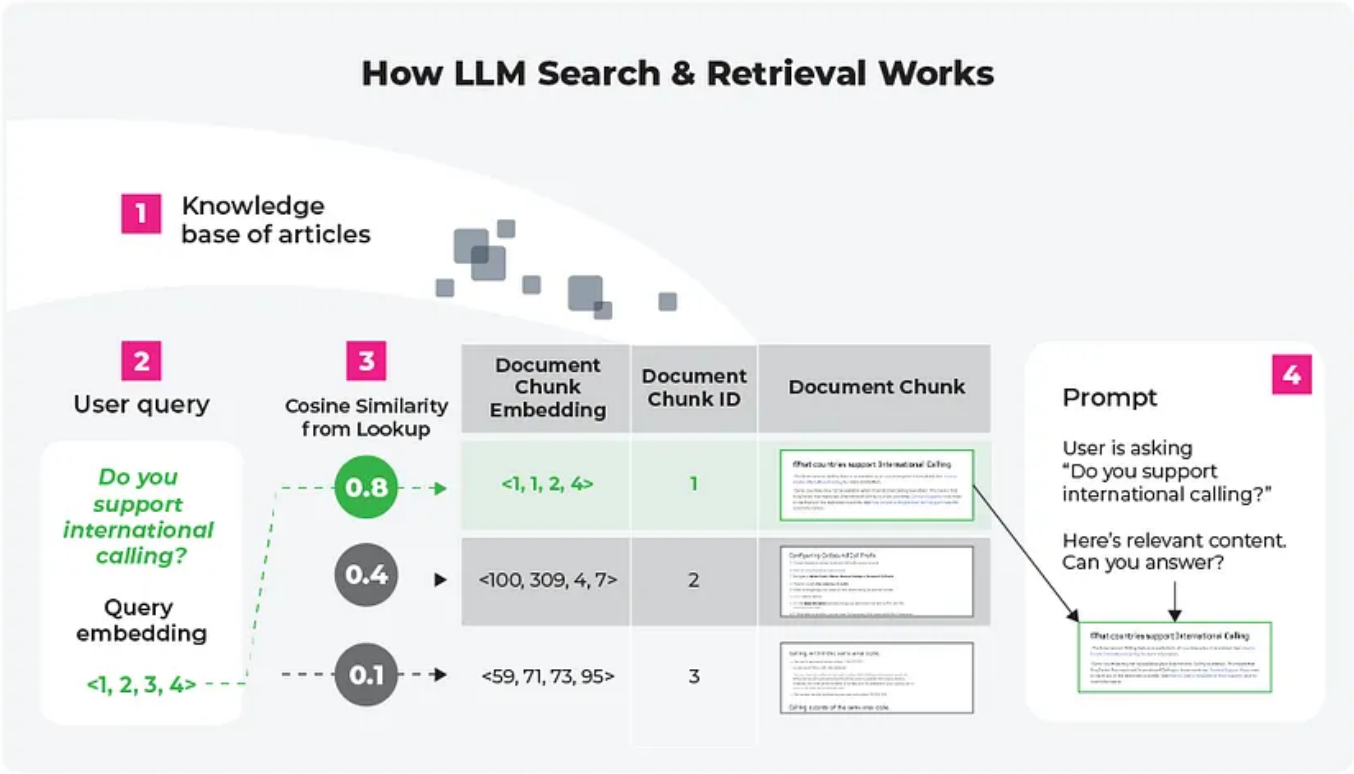


Image By Arize

The picture above shows how search and retrieval is used with a prompt template ('4' in the above image) to generate a final LLM prompt context. The above view is the search and retrieval LLM use case in its simplest form: a document is broken into chunks, these chunks are embedded into a vector store, and the search and retrieval process pulls on this context to shape LLM output.

This approach offers several advantages. First, it significantly reduces the time and computational resources required for the LLM to process large

amounts of data, as it only needs to interact with the relevant chunks instead of the entire documentation.

Second, it allows for real-time updates to the database. As product documentation evolves, the corresponding chunks in the vector database can be easily updated. This ensures that the chatbot always provides the most up-to-date information.

Finally, by focusing on semantically relevant chunks, the LLM can provide more precise and contextually appropriate responses, leading to improved customer satisfaction.

1.1. Issues with Search & Retrieval

While the search and retrieval method greatly enhances the efficiency and accuracy of LLMs, it's not without potential pitfalls. Identifying these issues early can prevent them from impacting user experience.

One such challenge arises when a user inputs a query that doesn't closely match any chunks in the vector store. The system looks for a needle in a haystack but finds no needle at all. This lack of match, often caused by unique or highly specific queries, can leave the system to draw on the "most similar" chunks available — ones that aren't entirely relevant.

In turn, this leads to a subpar response from the LLM. Since the LLM depends on the relevance of the chunks to generate responses, the lack of an appropriate match could result in an output that's tangentially related or even completely unrelated to the user's query.

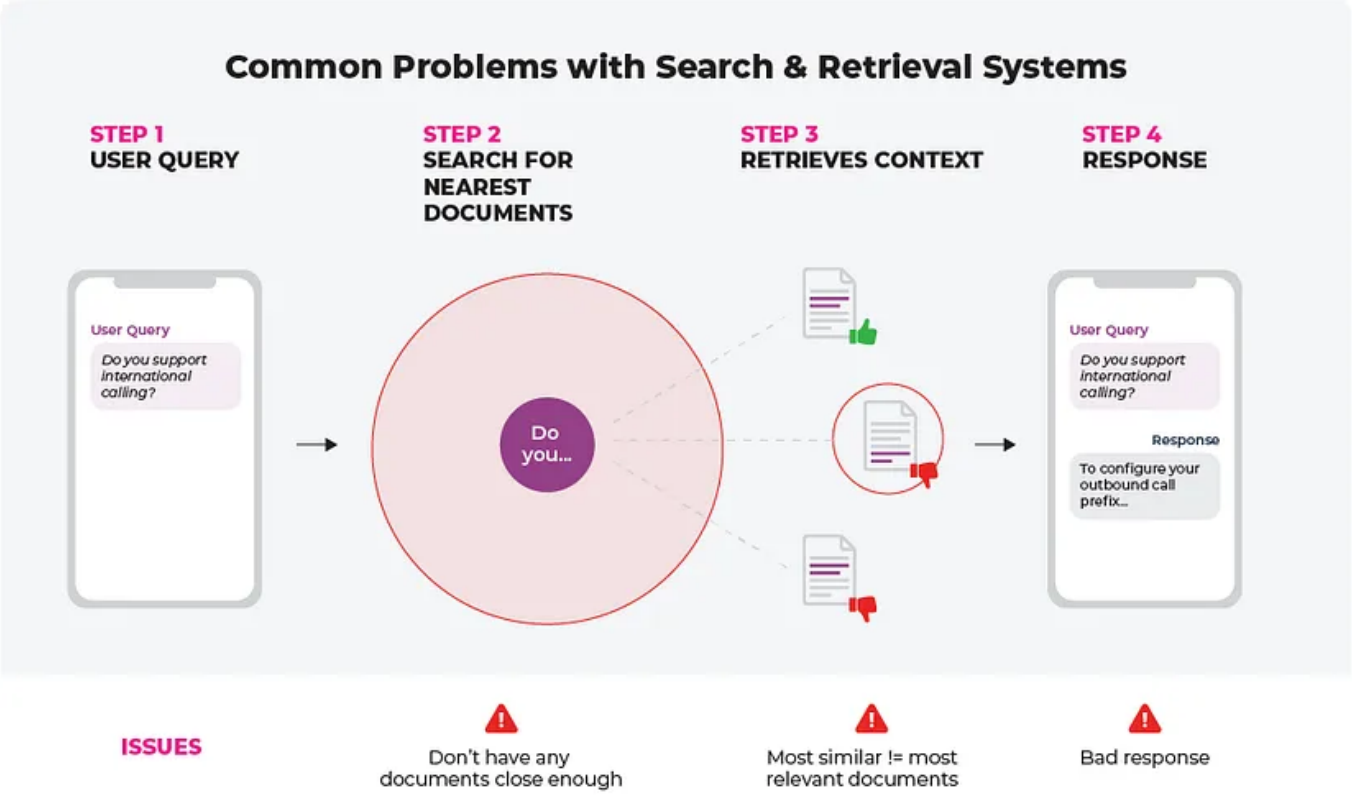


Image By Arize

Irrelevant or subpar responses from the LLM can frustrate users, lowering their satisfaction and ultimately causing them to lose trust in the system and product as a whole.

Monitoring three main things can help prevent these issues:

Query Density (Drift): Query density refers to how well user queries are covered by the vector store. If query density drifts significantly, it signals that our vector store may not be capturing the full breadth of user queries, resulting in a shortage of closely associated chunks. Regularly monitoring query density enables us to spot these gaps or shortcomings. With this insight, we can augment the vector store by incorporating more relevant chunks or refining the existing ones, improving the system's ability to fetch data in response to user queries.

Ranking Metrics: These metrics evaluate how well the search and retrieval system is performing in terms of selecting the most relevant chunks. If the ranking metrics indicate a decline in performance, it's a signal that the system's ability to distinguish between relevant and irrelevant chunks might need refinement.

User Feedback: Encouraging users to provide feedback on the quality and relevance of the LLM's responses helps gauge user satisfaction and identify areas for improvement. Regular analysis of this feedback can point out patterns and trends, which can then be used to adjust your application as necessary.

1.2. Optimizing Search & Retrieval

Optimization of search and retrieval processes should be a constant endeavor throughout the lifecycle of your LLM-powered application, from the building phase through to post-production.

During the building phase, attention should be given to developing a robust testing and evaluation strategy. This approach allows us to identify potential issues early on and optimize our strategies, forming a solid foundation for the system.

Key areas to focus on include:

- **Chunking Strategy**: Evaluating how information is broken down and processed during this stage can help highlight areas for improvement in performance.
- **Retrieval Performance:** Assessing how well the system retrieves information can indicate if we need to employ different tools or

strategies, such as context ranking or HYDE.

Upon release, optimization efforts should continue as we enter the post-production phase. Even after launch, with a well-defined evaluation strategy, we can proactively identify any emerging issues and continue to improve our model's performance. Consider approaches like:

- **Expanding our Knowledge Base**: Adding documentation can significantly improve our system's response quality. An expanded data set allows our LLM to provide more accurate and tailored responses.
- **Refining Chunking Strategy**: Further modifying the way information is broken down and processed can lead to marked improvements.
- **Enhancing Context Understanding**: Incorporating an extra 'context evaluation' step helps the system incorporate the most relevant context into the LLM's response.

Specifics on these and other strategies for continuous optimization will be detailed in the following sections of this course. Remember, the goal is to create a system that not only meets users' needs at launch but also evolves with them over time.

2. Types of Search

We have to remember that vector databases are not the panacea of search — they are very good at *semantic* search, but in many cases, traditional keyword search can yield more relevant results and increase user satisfaction. Why is that? It's largely to do with the fact that ranking based on metrics like cosine similarity causes results that have a higher similarity score to appear above

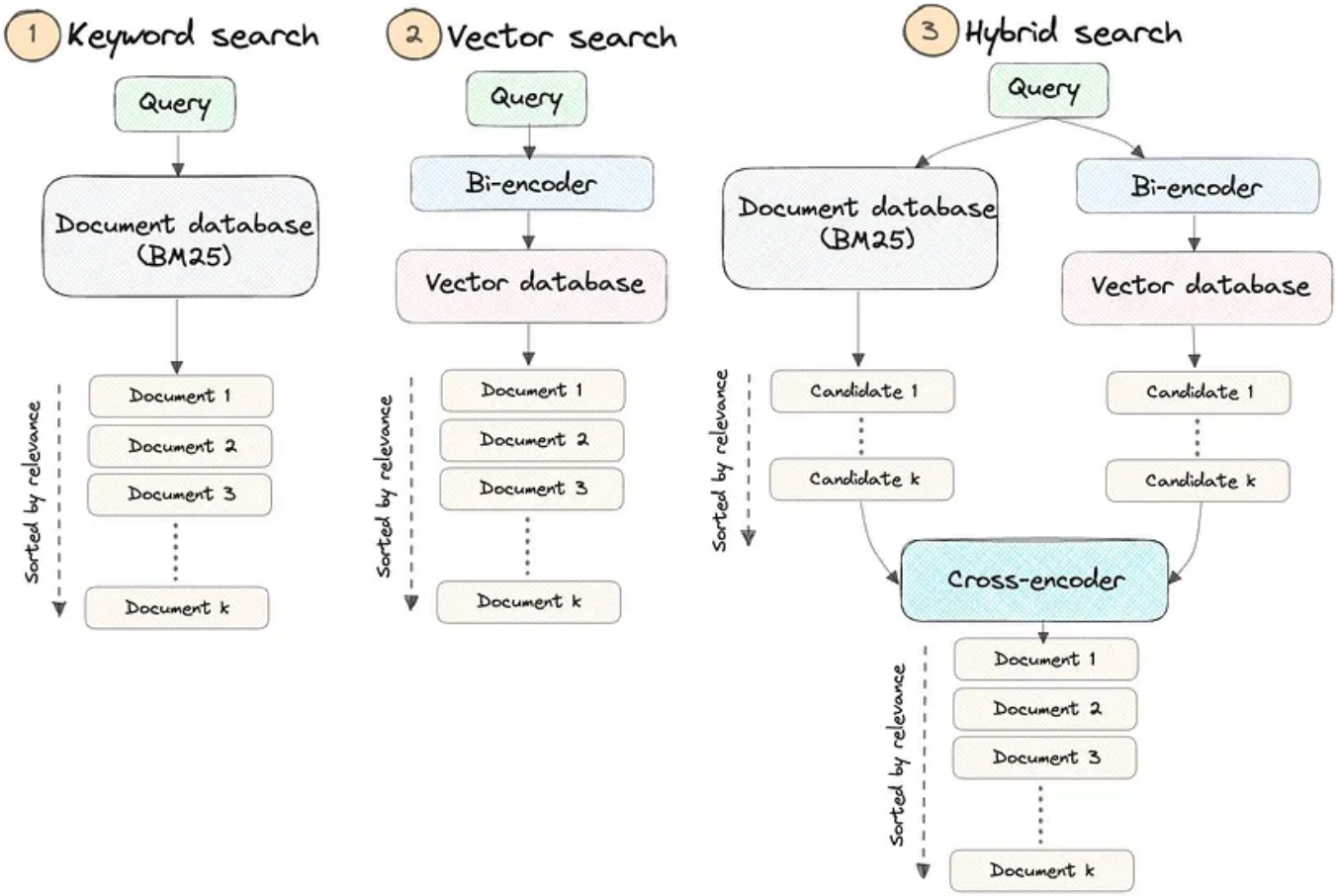
partial matches that may contain specific input keywords, reducing their relevance to the end user.

However, pure keyword search also has its limitations — in case the user enters a term that is semantically similar to the stored data (but is not exact), potentially useful and relevant results are not returned. As a result of this trade-off, real-world use cases for search & retrieval demand a combination of keyword and vector searches, **of which vector databases form a key component** (because they house the embeddings, enabling semantic similarity search and can scale to very large datasets).

To summarize the points above:

- **Keyword search**: Finds relevant, useful results when the user *knows* what they're looking for and expects results that match exact phrases in their search terms. Does **not** require vector databases.
- **Vector search**: Finds relevant results when the user *doesn't* know what exactly they're looking for. Requires a vector database.
- **Hybrid (keyword + vector) search**: Typically combines candidate results from full-text keyword and vector searches and re-ranks them using cross-encoder models. Requires both a document database and a vector database.

This can be effectively visualized per the diagram below:



Search

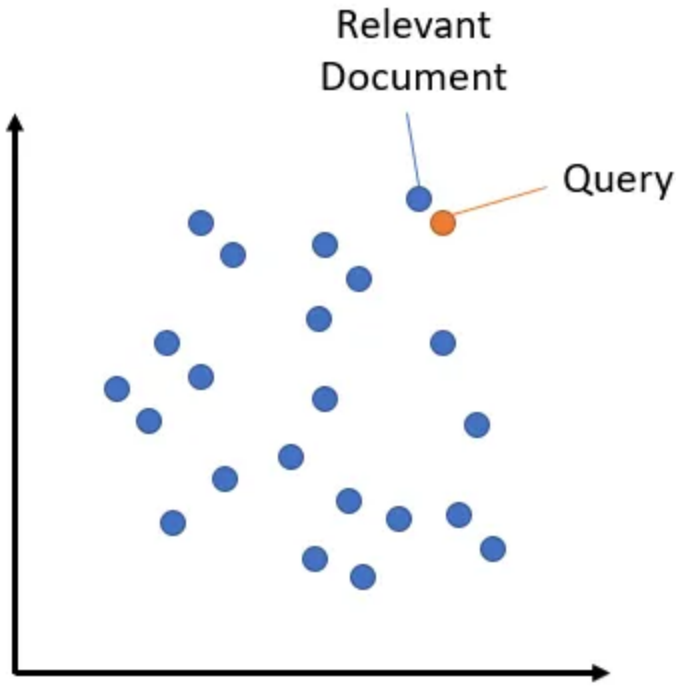
2.1. Semantic Search

Semantic search seeks to improve search accuracy by understanding the content of the search query. In contrast to traditional search engines which only find documents based on lexical matches, semantic search can also find synonyms.

2.1.1. Background

The idea behind semantic search is to embed all entries in our corpus, whether they be sentences, paragraphs, or documents, into a vector space.

At search time, the query is embedded into the same vector space and the closest embeddings from our corpus are found. These entries should have a high semantic overlap with the query.



2.2. Symmetric vs. Asymmetric Semantic Search

A **critical distinction** for our setup is *symmetric* vs. *asymmetric semantic search*:

- For **symmetric semantic search,** our query and the entries in our corpus are of about the same length and have the same amount of content. An example would be searching for similar questions: Our query could for example be *"How to learn Python online?"* and we want to find an entry like *"How to learn Python on the web?"*. For symmetric tasks, we could potentially flip the query and the entries in our corpus.
- For **asymmetric semantic search**, we usually have a **short query** (like a question or some keywords), and we want to find a longer paragraph answering the query. An example would be a query like *"What is Python"* and we wand to find the paragraph *"Python is an interpreted, high-level and general-purpose programming language. Python's design philosophy …"*. For asymmetric tasks, flipping the query and the entries in our corpus usually does not make sense.

We must choose **the right model** for our type of task.

Suitable models for **symmetric semantic search**: Pre-Trained Sentence Embedding Models

Suitable models for **asymmetric semantic search**: Pre-Trained MS MARCO Models

3. Retrieval Algorithms

3.1. Similarity Search (Vanilla Search) & Maximum Marginal Relevance(MMR)

When it comes to retrieving documents, the majority of methods will do a similarity metric like cosine similarity, euclidean distance, or dot product. All of these will return documents that are most similar to our query/question.

However, what if we want similar documents that are also diverse from each other? That is where Maximum Marginal Relevance (MMR) steps in.

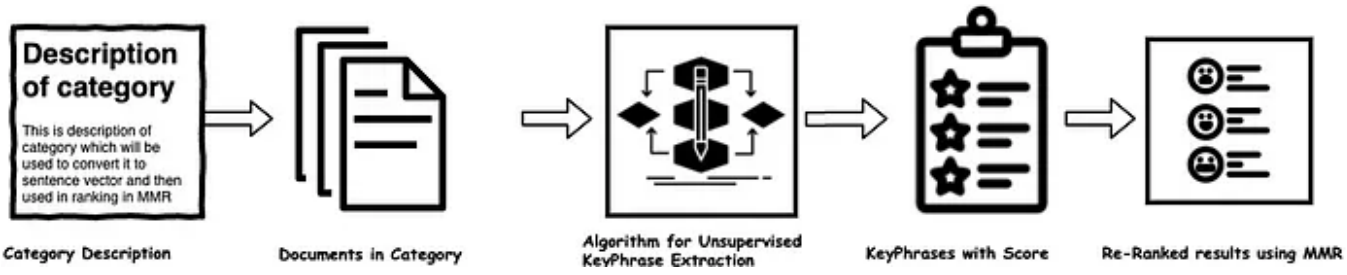
The goal is to take into account how similar retrieved documents *are to each other* when determining which to return. In theory, we should have a well-rounded, diverse set of documents.

In case of **unsupervised learning**, let's say our final keyPhrases are ranked like `Good Product, Great Product, Nice Product, Excellent Product, Easy Install, Nice UI, Light weight etc.` But there is an issue with this approach, all the phrases like `good product, nice product, excellent product` are similar and define the same property of the product and are ranked higher.

Suppose we have a space to show just 5 key phrases, in that case, we don't want to show all these similar phrases.

We want to properly utilize this limited space such that the information displayed by the Keyphrases about the documents is diverse enough. Similar types of phrases should not dominate the whole space and users can see a variety of information about the document.

1. **Remove redundant phrases using cosine similarity**
2. **Re-ranking the key phrases using MMR**



MMR

Above are two widely used Retrieval methods. Other methods like *Multi Query Retrieval, Long-Context Reorder, Multi-Vector Retriever, Parent Document Retriever, Self-Querying, Time-weighted Vector Store Retrieval* are some of the advanced Retrieval strategies that we will cover in a separate blog post.

Now let's discuss the Retrieval and Re-ranking pipeline below and let's see how it enhances the results.

4. Retrieve & Re-Rank

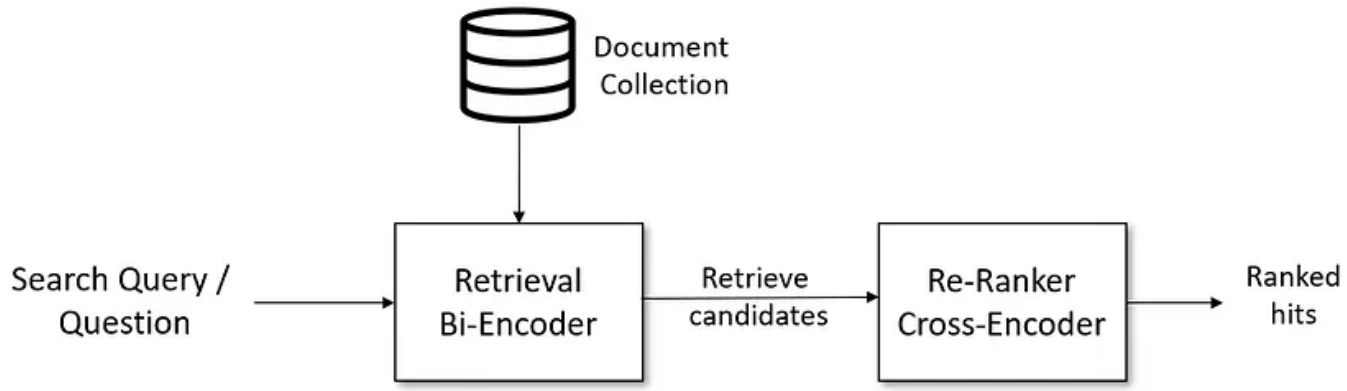
In Semantic Search we have shown how to use Sentence Transformer to compute embeddings for queries, sentences, and paragraphs and how to use

this for semantic search.

For complex search tasks, for example, for question-answering retrieval, the search can significantly be improved by using **Retrieve & Re-Rank**.

4.1. Retrieve & Re-Rank Pipeline

A pipeline for information retrieval / question-answering retrieval that works well is the following. All components are provided and explained in this article:



By SBERT

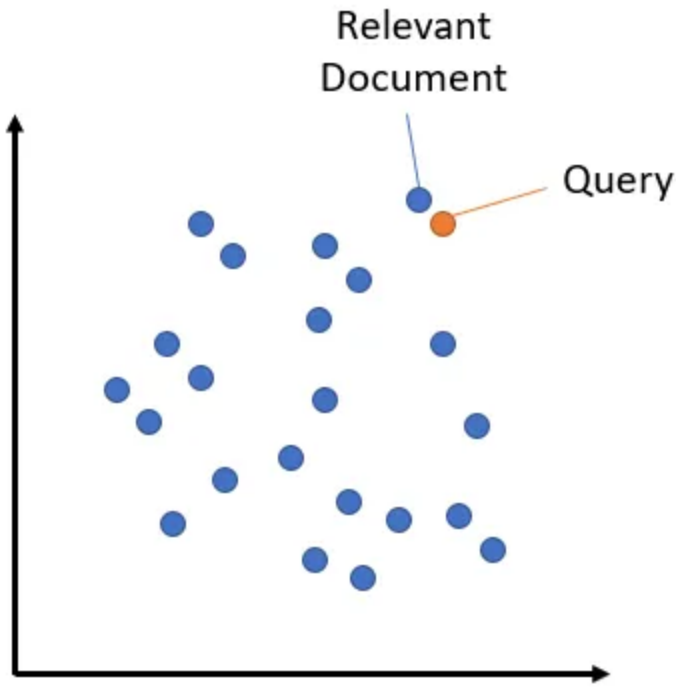
Given a search query, we first use a **retrieval system** that retrieves a large list of e.g. 100 possible hits that are potentially relevant for the query. For the retrieval, we can use either lexical search, e.g. with ElasticSearch, or we can use dense retrieval with a bi-encoder.

However, the retrieval system might retrieve documents that are not that relevant to the search query. Hence, in the second stage, we use a **re-ranker** based on a **cross-encoder** that scores the relevancy of all candidates for the given search query.

The output will be a ranked list of hits we can present to the user.

4.2. Retrieval: Bi-Encoder

Lexical search looks for literal matches of the query words in our document collection. It will not recognize synonyms, acronyms or spelling variations. In contrast, semantic search (or dense retrieval) encodes the search query into vector space and retrieves the document embeddings that are close in vector space.



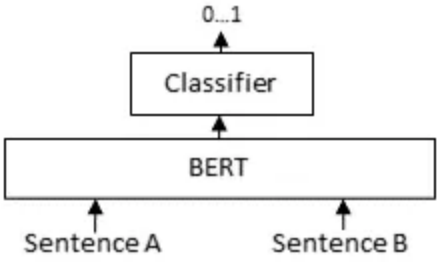
Semantic search overcomes the shortcomings of lexical search and can recognize synonyms and acronyms. Have a look at the semantic search article for different options to implement semantic search.

4.3. Re-Ranker: Cross-Encoder

The retriever has to be efficient for large document collections with millions of entries. However, it might return irrelevant candidates.

A re-ranker based on a Cross-Encoder can substantially improve the final results for the user. The query and a possible document are passed

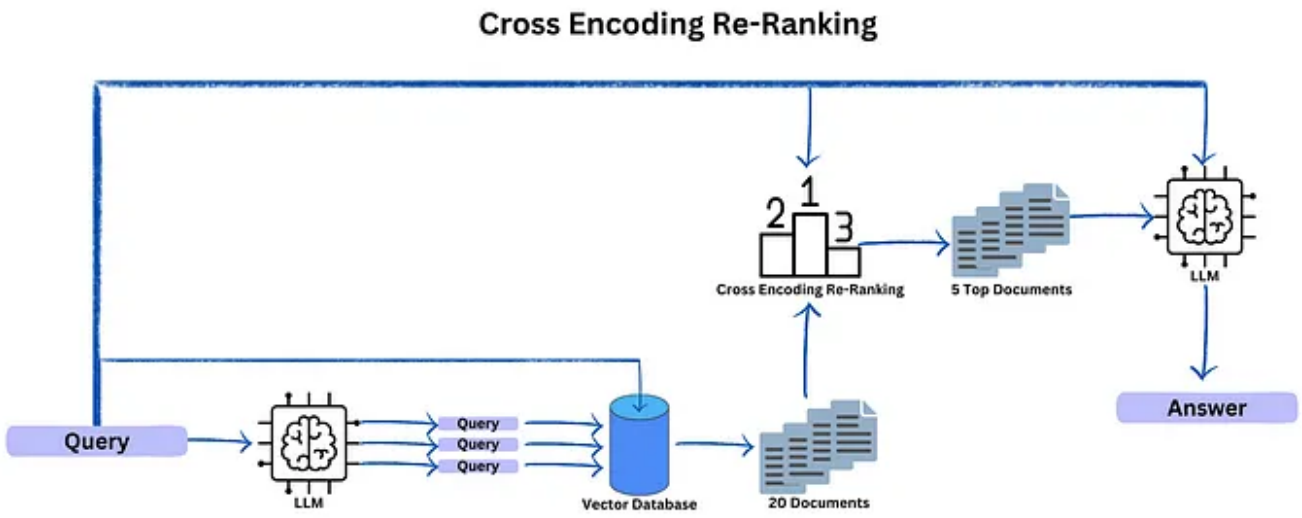
simultaneously to the transformer network, which then outputs a single score between 0 and 1 indicating how relevant the document is for the given query.



By SBERT

The advantage of Cross-Encoders is the higher performance, as they perform attention across the query and the document.

Scoring thousands or millions of (query, document)-pairs would be rather slow. Hence, we use the retriever to create a set of e.g. 100 possible candidates which are then re-ranked by the Cross-Encoder.



Cross Encoding Re-Ranking High-Level Flow

4.4. Example Scripts

- **retrieve_rerank_simple_wikipedia.ipynb** [Colab Version]: This script uses the smaller Simple English Wikipedia as a document collection to provide answers to user questions/search queries. First, we split all Wikipedia articles into paragraphs and encode them with a bi-encoder. If a new query/question is entered, it is encoded by the same bi-encoder and the paragraphs with the highest cosine-similarity are retrieved (see semantic search). Next, the retrieved candidates are scored by a Cross-Encoder re-ranker and the 5 passages with the highest score from the Cross-Encoder are presented to the user.
- **in_document_search_crossencoder.py:** If have only have a small set of paragraphs, we don't have the retrieval stage. This is for example the case if we want to perform a search within a single document. In this example, take the Wikipedia article about Europe and split it into paragraphs. Then, the search query/question and all paragraphs are

scored using the Cross-Encoder re-ranker. The most relevant passages for the query are returned.

4.5. Pre-trained Bi-Encoders (Retrieval)

The bi-encoder produces embeddings independently for our paragraphs and our search queries. We can use it like this:

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('model_name')

docs = ["My first paragraph. That contains information", "Python is a programmin
document_embeddings = model.encode(docs)

query = "What is Python?"
query_embedding = model.encode(query)
```

For more details on how to compare the embeddings, please visit semantic search.

We provide pre-trained models based on:

- **MS MARCO:** 500k real user queries from Bing search engine. See MS MARCO models

4.6. Pre-trained Cross-Encoders (Re-Ranker)

For pre-trained models, we can refer: MS MARCO Cross-Encoders

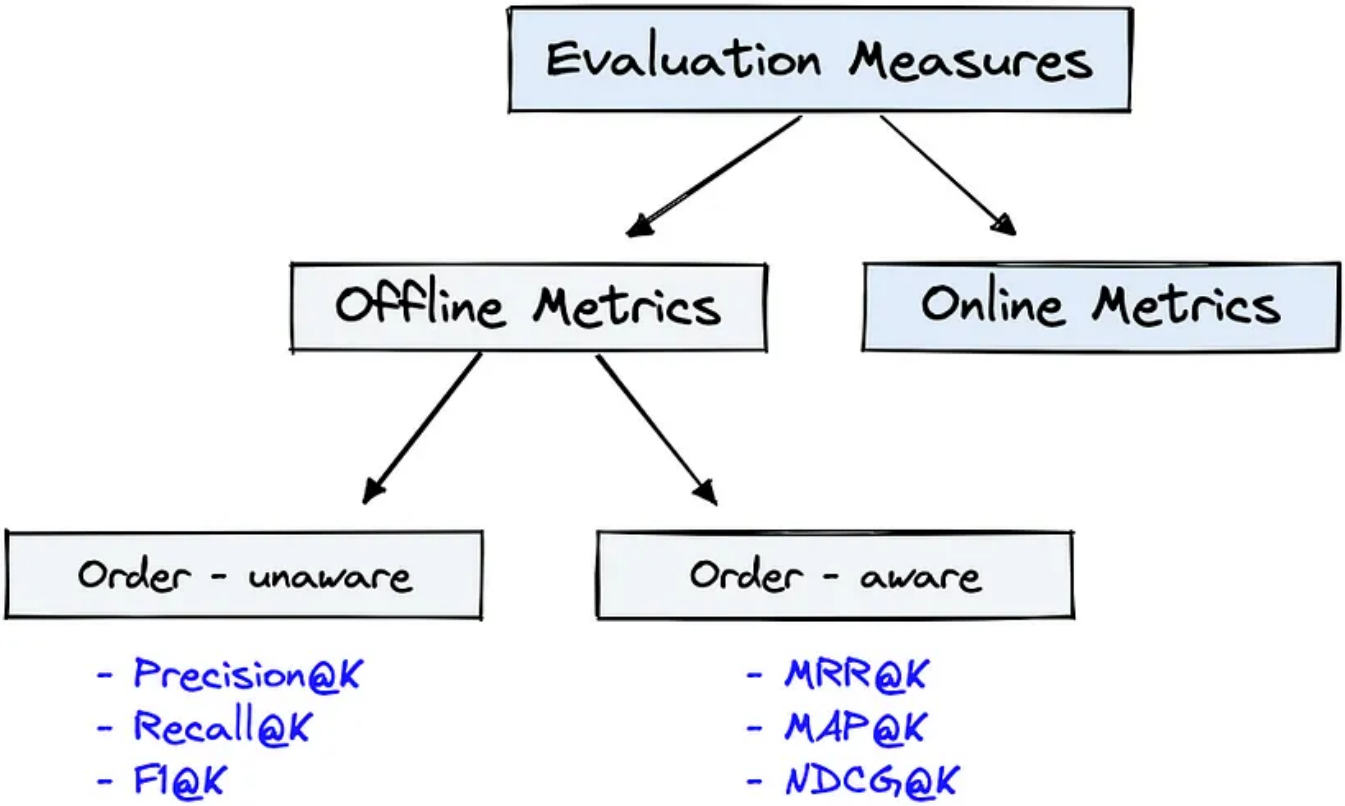
5. Evaluation of Information Retrieval

Evaluation of information retrieval (IR) systems is critical to making well-informed design decisions. From search to recommendations, evaluation measures are paramount to understanding what does and does not work in retrieval.

Evaluation measures for IR systems can be split into *two* categories: *online* or *offline* metrics.

Online metrics are captured during actual usage of the IR system when it is *online*. These consider user interactions like whether a user clicked on a recommended show from Netflix or if a particular link was clicked from an email advertisement (the click-through rate or CTR). There are many online metrics, but they all relate to some form of user interaction.

Offline metrics are measured in an isolated environment before deploying a new IR system. These look at whether a particular set of *relevant* results are returned when retrieving items with the system.



By Pinecone

Evaluation measures can be categorized as either offline or online metrics. Offline metrics can be further divided into order-unaware or order-aware, which we will explain soon.

Organizations often use *both* offline and online metrics to measure the performance of their IR systems. It begins, however, with offline metrics to predict the system's performance *before deployment*.

We will focus on the most useful and popular offline metrics:

- Recall@K
- **M**ean **R**eciprocal **R**ank (MRR)
- **M**ean **A**verage **P**recision@K (MAP@K)
- **N**ormalized **D**iscounted **C**umulative **G**ain (NDCG@K)

These metrics are deceptively simple yet provide invaluable insight into the performance of IR systems.

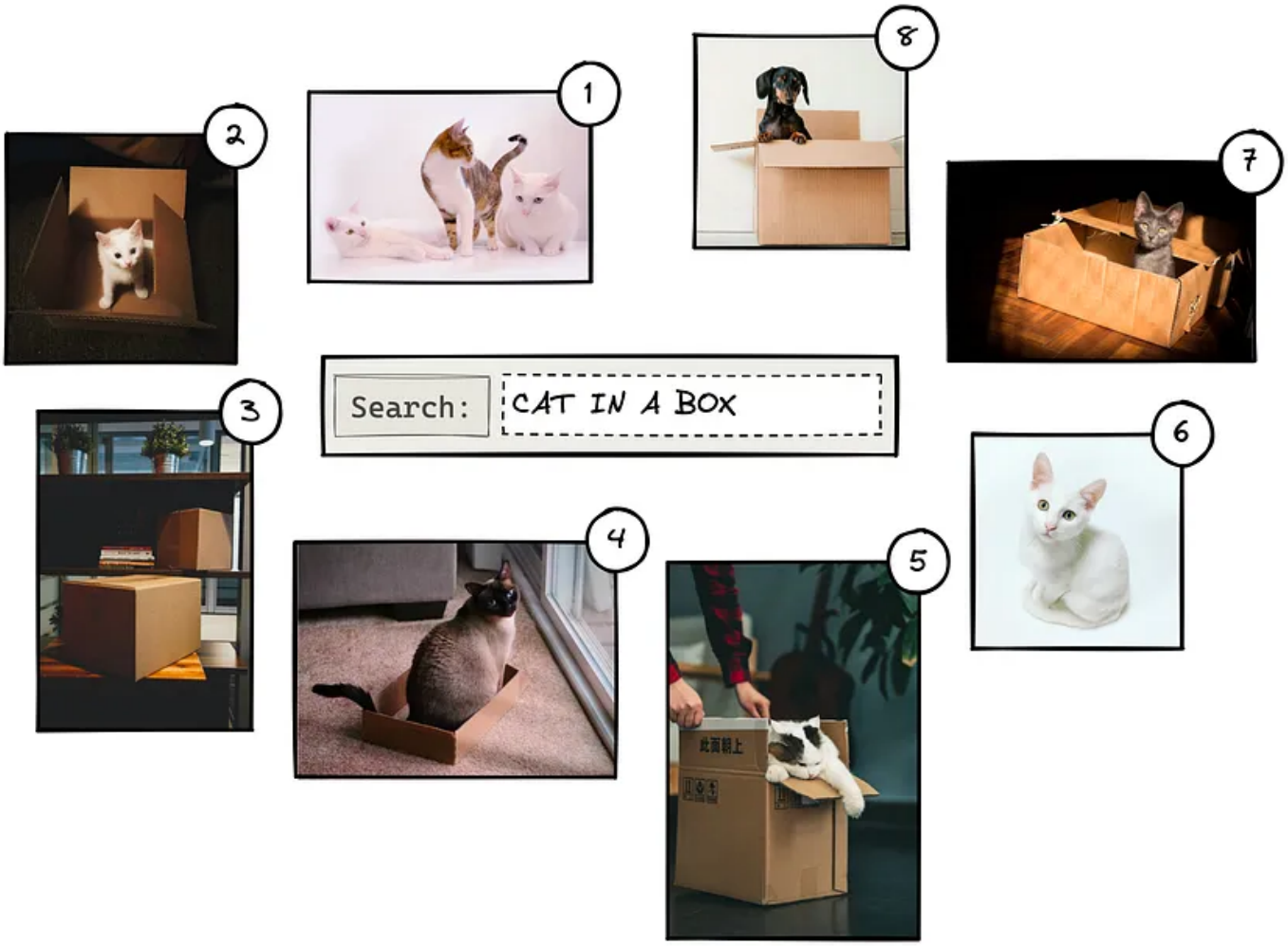
We can use one or more of these metrics in different evaluation stages. During the development of Spotify's podcast search; *Recall@K* (using *K*=1) was used during training on "evaluation batches", and after training, both *Recall@K* and *MRR* (using *K*=30) were used with a much larger evaluation set.

For now, understand that Spotify was able to predict system performance *before* deploying anything to customers. This allowed them to deploy successful A/B tests and significantly increase podcast engagement.

We have two more subdivisions for these metrics; *order-aware* and *order-unaware*. This refers to whether the order of results impacts the metric score. If so, the metric is *order-aware*. Otherwise, it is *order-unaware*.

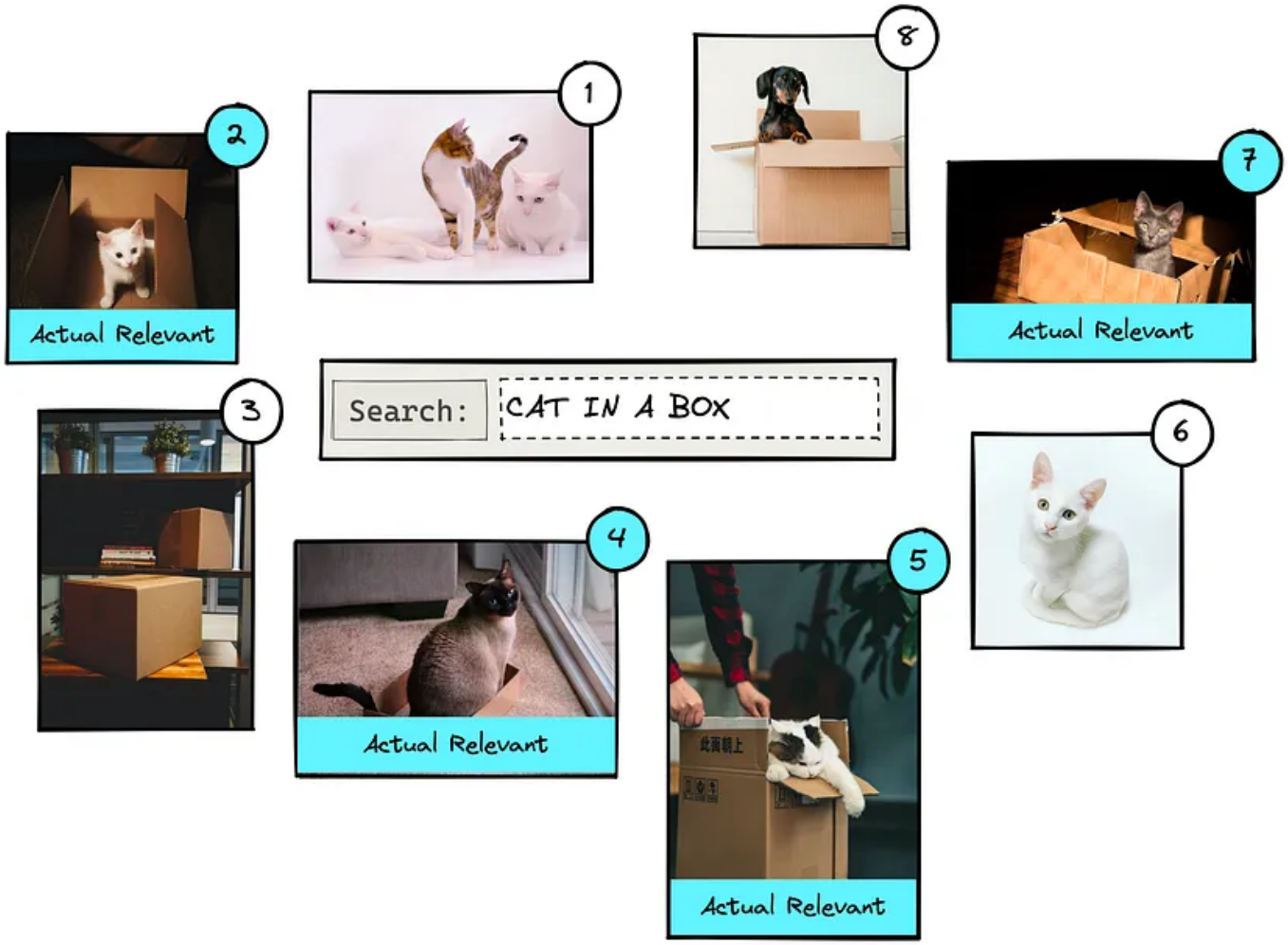
5.1. Example

Throughout the article, we will be using a *very* small dataset of eight images. In reality, this number is likely to be millions or more.



Example query and ranking of the eight possible results.

If we were to search for *"cat in a box"*, we may return something like the above. The numbers represent the relevance *rank* of each image as predicted by the IR system. Other queries would yield a different order of results.



Example query and ranking with actual relevant results highlighted.

We can see that results *2*, *4*, *5*, and *7* are *actual relevant* results. The other results are *not* relevant as they show cats *without* boxes, boxes *without* cats, or a dog.

5.2. Actual vs. Predicted

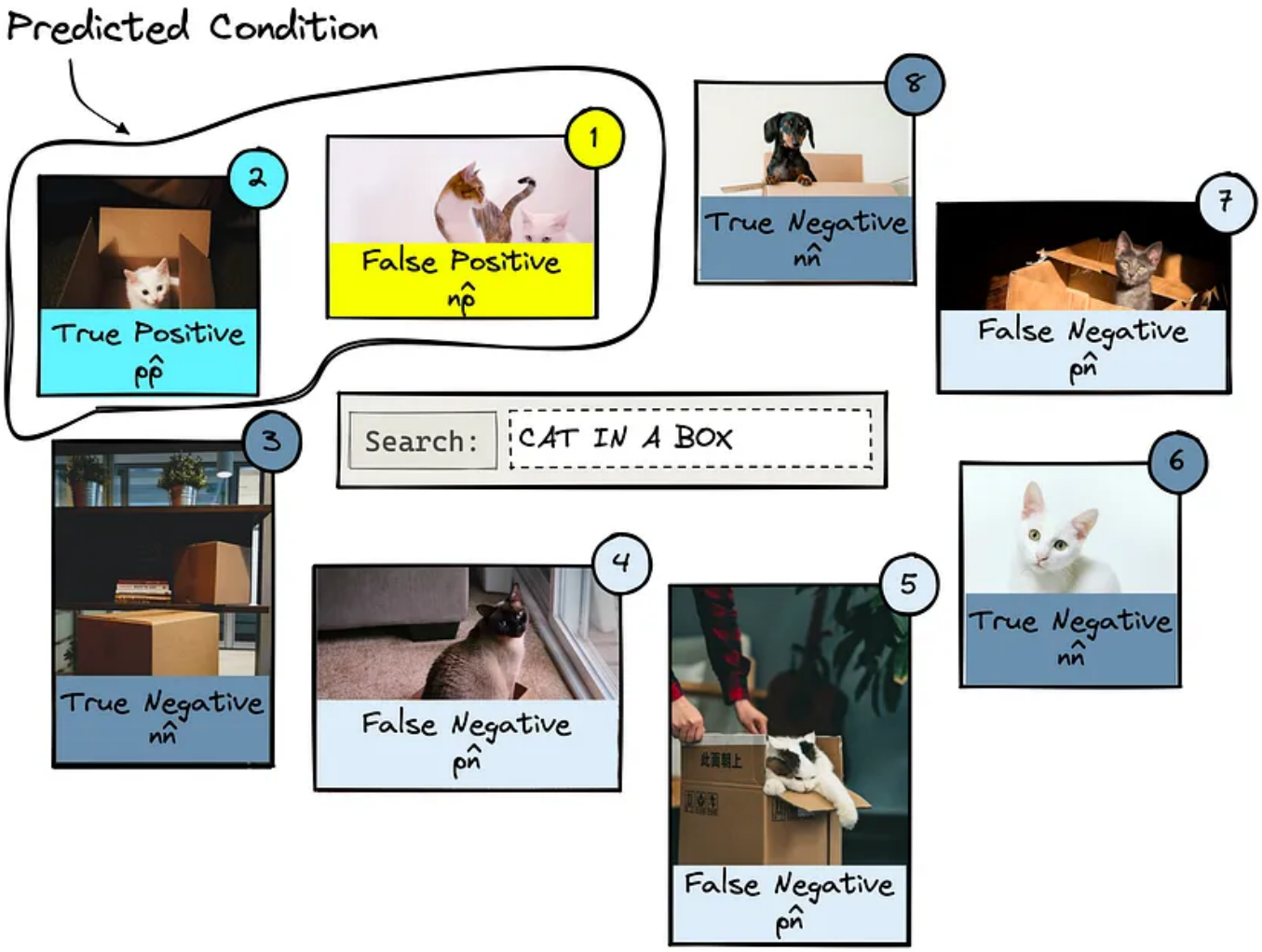
When evaluating the performance of the IR system, we will be comparing *actual* vs. *predicted* conditions, where:

- **Actual condition** refers to the true label of every item in the dataset. These are *positive* (*p*) if an item is relevant to a query or *negative* (*n*) if an item is *ir*relevant to a query.

- **Predicted condition** is the *predicted* label returned by the IR system. If an item is returned, it is predicted as being *positive* (*p^*) and, if it is not returned, is predicted as a *negative* (*n^*).

From these actual and predicted conditions, we create a set of outputs from which we calculate all of our offline metrics. Those are the true/false positives and true/false negatives.

The *positive* results focus on what the IR system returns. Given our dataset, we ask the IR system to return *two* items using the *"cat in a box"* query. If it returns an *actual relevant* result this is a *true positive* (*pp^*); if it returns an irrelevant result, we have a *false positive* (*np^*).



By Pinecone

For *negative* results, we must look at what the IR system *does not* return. Let's query for two results. Anything that *is relevant* but is *not* returned is a *false negative* (*pn*^). Irrelevant items that were *not* returned are *true negatives* (*nn*^).

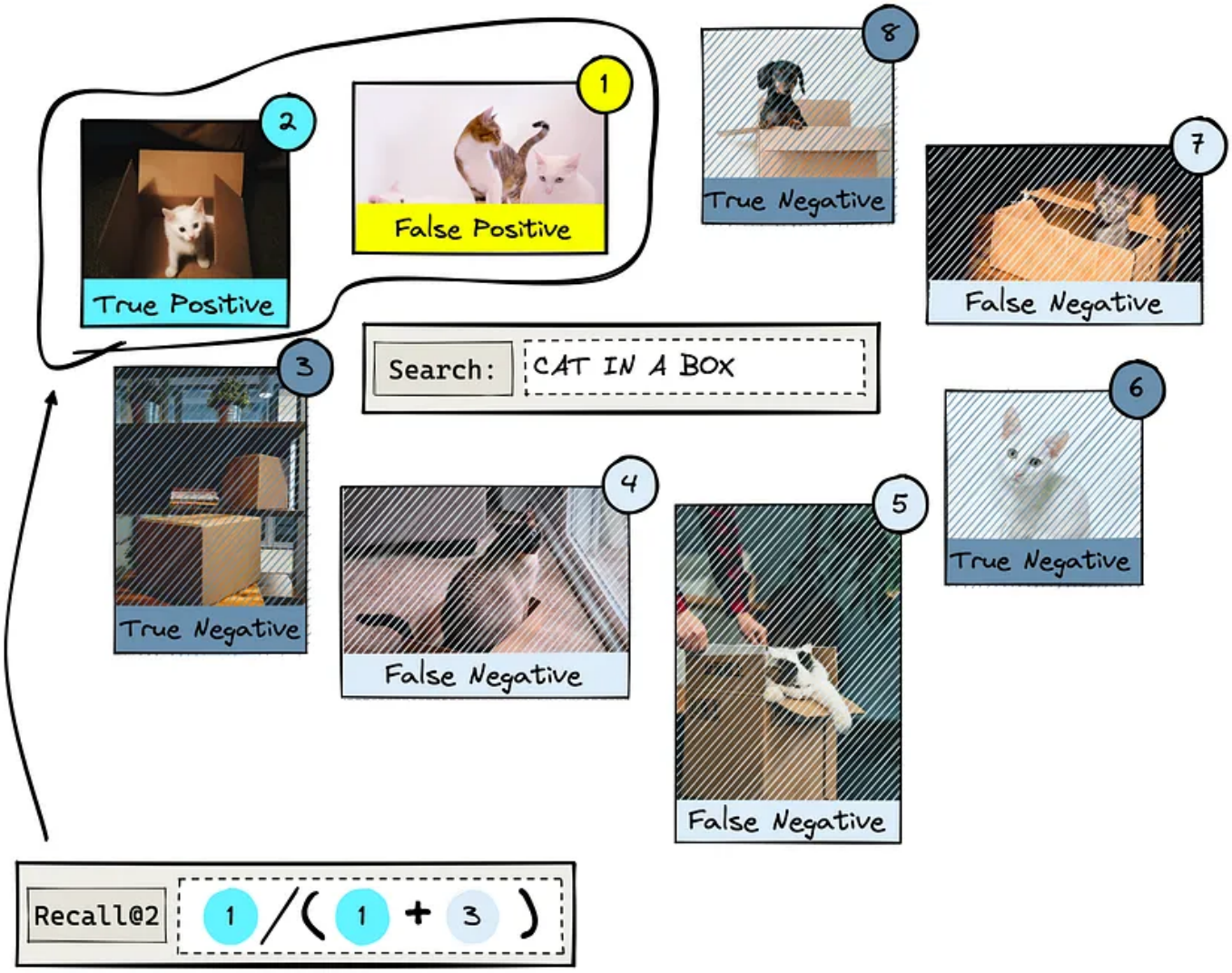
With all of this in mind, we can begin with the first metric.

5.3. Recall@K

Recall@K is one of the most interpretable and popular offline evaluation metrics. It measures how many relevant items were returned (*pp*^) against how many relevant items exist in the entire dataset (*pp*^+*pn*^).

$$Recall@K = \frac{truePositives}{truePositives + falseNegatives} = \frac{p\hat{p}}{p\hat{p} + p\hat{n}}$$

The *K* in this and all other offline metrics refers to the number of items returned by the IR system. In our example, we have a total number of $N = 8$ items in the entire dataset, so *K* can be any value between [1 ,..., *N*].

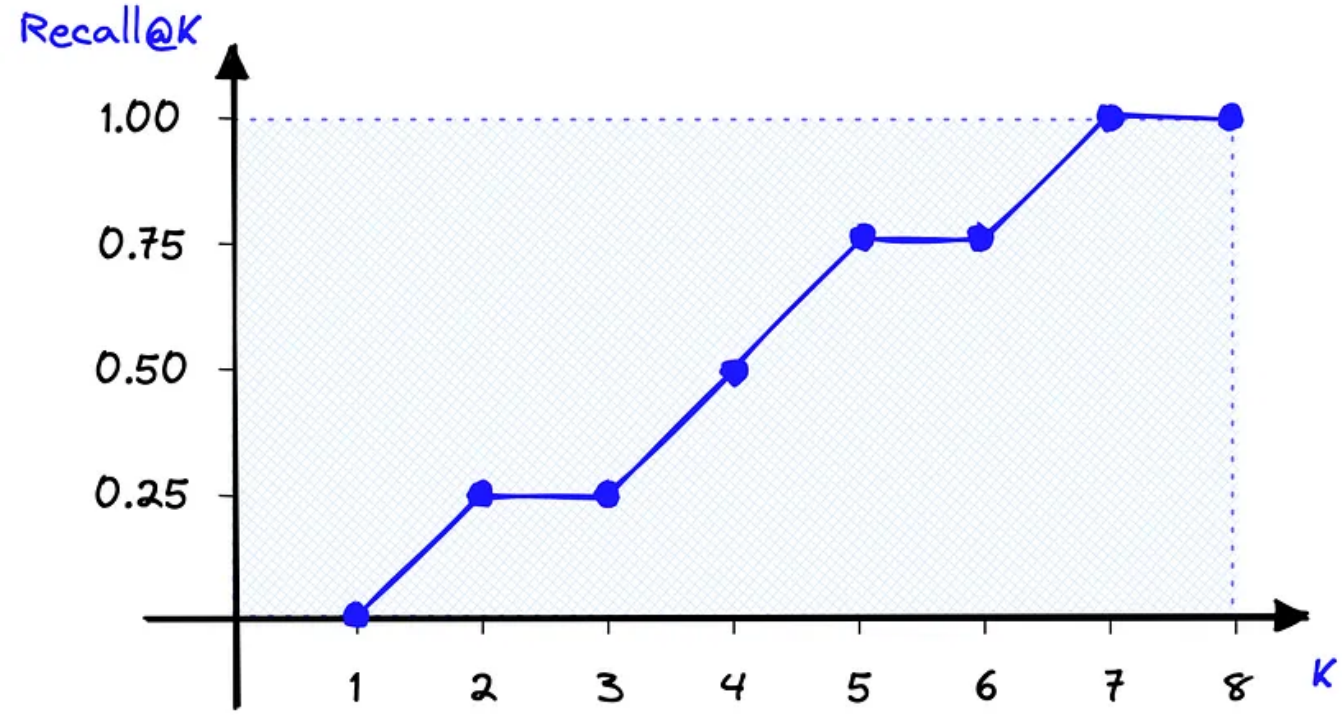


With recall@2 we return the predicted top K = 2 most relevant results.

When $K = 2$, our *recall@2* score is calculated as the number of *returned* relevant results over the total number of relevant results in the *entire dataset*. That is:

$$Recall@2 = \frac{p\hat{p}}{p\hat{p} + p\hat{n}} = \frac{1}{1+3} = 0.25$$

With recall@K, the score improves as K increases and the scope of returned items increases.



With recall@K we will see the score increase as K increases and more positives (whether true or false) are returned.

We can calculate the same recall@K score easily in Python. For this, we will define a function named **recall** that takes lists of *actual conditions* and *predicted conditions*, a *K* value, and returns a recall@K score.

```
# recall@k function
def recall(actual, predicted, k):
    act_set = set(actual)
    pred_set = set(predicted[:k])
    result = round(len(act_set & pred_set) / float(len(act_set)), 2)
    return result
```

Using this, we will replicate our eight-image dataset with *actual relevant* results in rank positions *2*, *4*, *5*, and *7*.

```
actual = ["2", "4", "5", "7"]
predicted = ["1", "2", "3", "4", "5", "6", "7", "8"]
for k in range(1, 9):
    print(f"Recall@{k} = {recall(actual, predicted, k)}")
```

Output :

```
Recall@1 = 0.0
Recall@2 = 0.25
Recall@3 = 0.25
Recall@4 = 0.5
Recall@5 = 0.75
Recall@6 = 0.75
Recall@7 = 1.0
Recall@8 = 1.0
```

Pros and Cons

Recall@K is undoubtedly one of the most easily interpretable evaluation metrics. We know that a perfect score indicates that all relevant items are being returned. We also know that a smaller *k* value makes it harder for the IR system to score well with recall@K.

Still, there are disadvantages to using *recall@K*. By increasing *K* to *N* or near *N*, we can return a perfect score every time, so relying solely on recall@K can be deceptive.

Another problem is that it is an *order-unaware metric*. That means if we used recall@4 and returned one relevant result at rank *one*, we would score the same as if we returned the same result at rank *four*. Clearly, it is better to

return the actual relevant result at a higher rank, but recall@K *cannot* account for this.

5.4. Mean Reciprocal Rank (MRR)

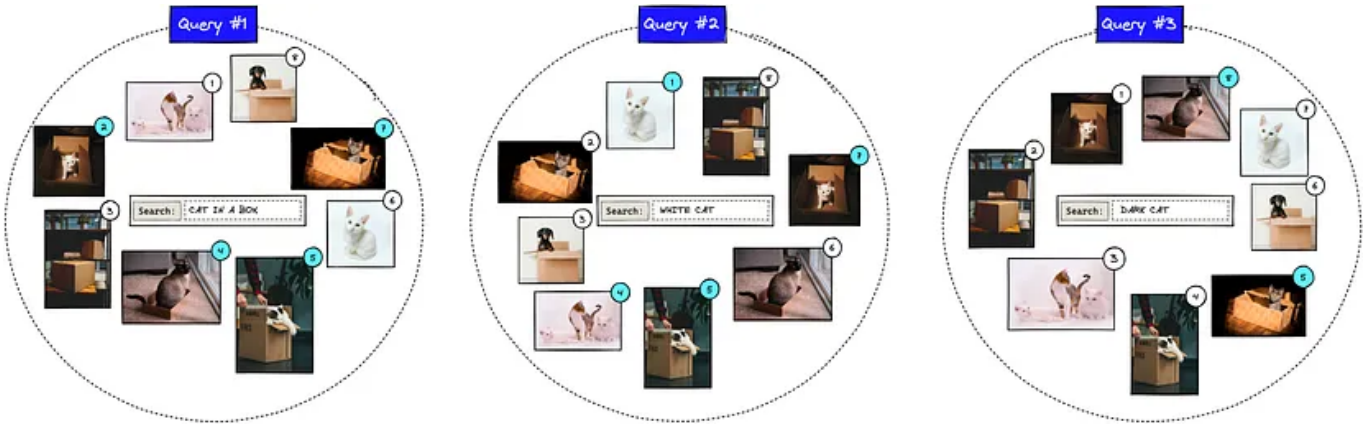
The **M**ean **R**eciprocal **R**ank (MRR) is an *order-aware metric*, which means that, unlike recall@K, returning an actual relevant result at rank *one* scores better than at rank *four*.

Another differentiator for MRR is that it is calculated based on multiple queries. It is calculated as:

$$RR = \frac{1}{Q}\sum_{q=1}^{Q}\frac{1}{rank_q}$$

Q is the number of queries, q is a specific query, and *rank-q* is the rank of the first *actual relevant* result for query q. We will explain the formula step-by-step.

Using our last example where a user searches for *"cat in a box"*. We add two more queries, giving us Q=3.



We perform three queries while calculating the MRR score.

We calculate the rank reciprocal 1/*rankq* for each query *q*. For the first query, the first actual relevant image is returned at position *two*, so the rank reciprocal is 1/2. Let's calculate the reciprocal rank for all queries:

$$query\ 1: \frac{1}{rank_1} = \frac{1}{2} = 0.5$$
$$query\ 2: \frac{1}{rank_2} = \frac{1}{1} = 1.0$$
$$query\ 3: \frac{1}{rank_3} = \frac{1}{5} = 0.2$$

Next, we sum all of these reciprocal ranks for queries *q*=[1,..., *Q*] (e.g., all three of our queries):

$$\sum_{q=1}^{Q} \frac{1}{rank_q} = 0.5 + 1.0 + 0.2 = 1.7$$

As we are calculating the **mean** reciprocal rank (**M**RR), we must take the average value by dividing our total reciprocal ranks by the number of queries *Q*:

$$MRR = \frac{1}{Q} \sum_{q=1}^{Q} \frac{1}{rank_q} = \frac{1}{3} 1.7 \cong 0.57$$

Now let's translate this into Python. We will replicate the same scenario where *Q*=3 using the same *actual relevant* results.

```
# relevant results for query #1, #2, and #3
actual_relevant = [
    [2, 4, 5, 7],
    [1, 4, 5, 7],
    [5, 8]
]
```

```
# number of queries
Q = len(actual_relevant)

# calculate the reciprocal of the first actual relevant rank
cumulative_reciprocal = 0
for i in range(Q):
    first_result = actual_relevant[i][0]
    reciprocal = 1 / first_result
    cumulative_reciprocal += reciprocal
    print(f"query #{i+1} = 1/{first_result} = {reciprocal}")
# calculate mrr
mrr = 1/Q * cumulative_reciprocal
# generate results
print("MRR =", round(mrr,2))
```

Output :

```
query #1 = 1/2 = 0.5
query #2 = 1/1 = 1.0
query #3 = 1/5 = 0.2
MRR = 0.57
```

And as expected, we calculate the same MRR score of *0.57*.

Pros and Cons

MRR has its own unique set of advantages and disadvantages. It is *order-aware*, a massive advantage for use cases where the rank of the first relevant result is important, like chatbots or question-answering.

On the other hand, we consider the rank of the *first* relevant item, but no others. That means for use cases where we'd like to return multiple items like recommendations or search engines, MRR is not a good metric. For example, if we'd like to recommend ~10 products to a user, we ask the IR system to retrieve 10 items. We could return just one *actual relevant* item in rank one and no other relevant items. Nine of ten irrelevant items is a terrible result, but MRR would score a perfect *1.0*.

Another *minor* disadvantage is that MRR is less readily interpretable compared to a simpler metric like recall@K. However, it is still more interpretable than many other evaluation metrics.

5.5. Mean Average Precision (MAP)

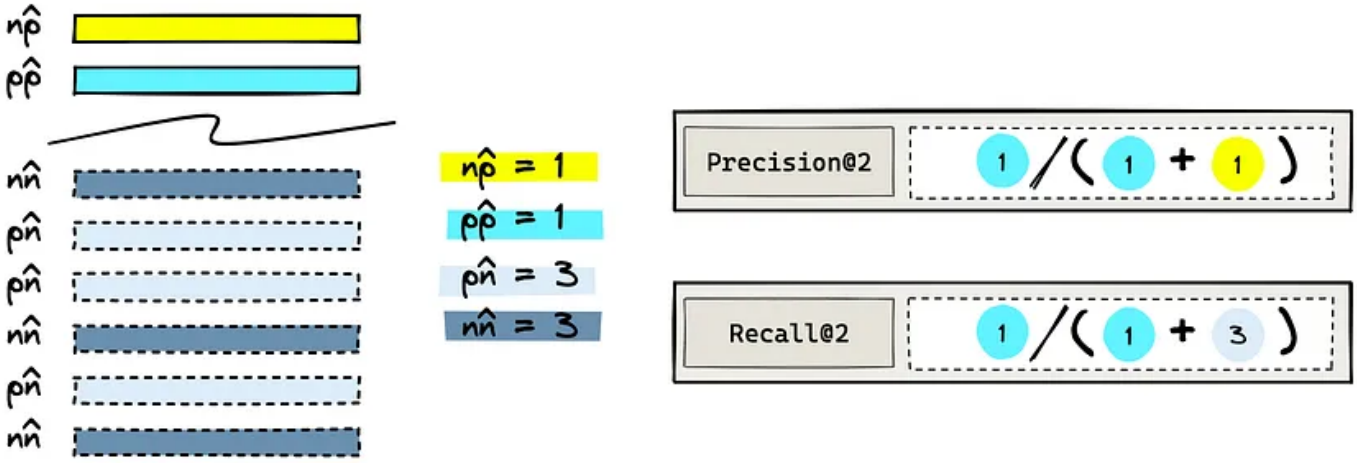
Mean **A**verage **P**recision@K (*MAP@K*) is another popular *order-aware* metric. At first, it seems to have an odd name, a *mean* of an *average?* It makes sense; we promise.

There are a few steps to calculating *MAP@K*. We start with another metric called *precision@K*:

$$Precision@K = \frac{truePositives}{truePositives + falsePositives} = \frac{p\hat{p}}{p\hat{p} + n\hat{p}}$$

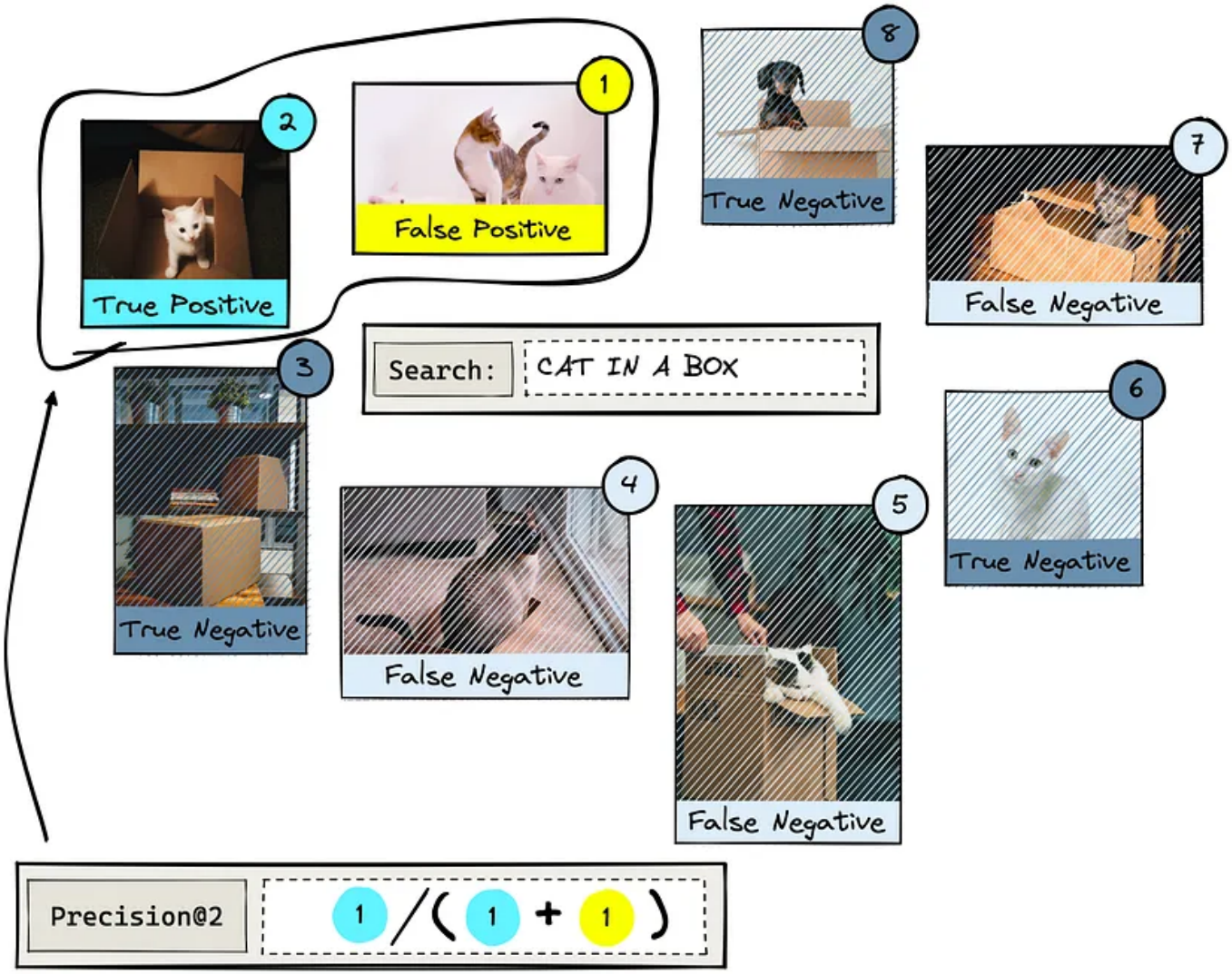
You may think this looks very similar to *recall@K*, and it is! The only difference is that we've swapped *pn^* in recall for *np^* here. That means we

now consider both actual relevant and non-relevant results *only* from the returned items. We *do not* consider non-returned items with *precision@K*.



The difference between Recall@K and Precision@K calculation where K = 2.

Let's return to the *"cat in a box"* example. We will return $K=2$ items and calculate *precision@2*.



Precision@K calculation where K = 2.

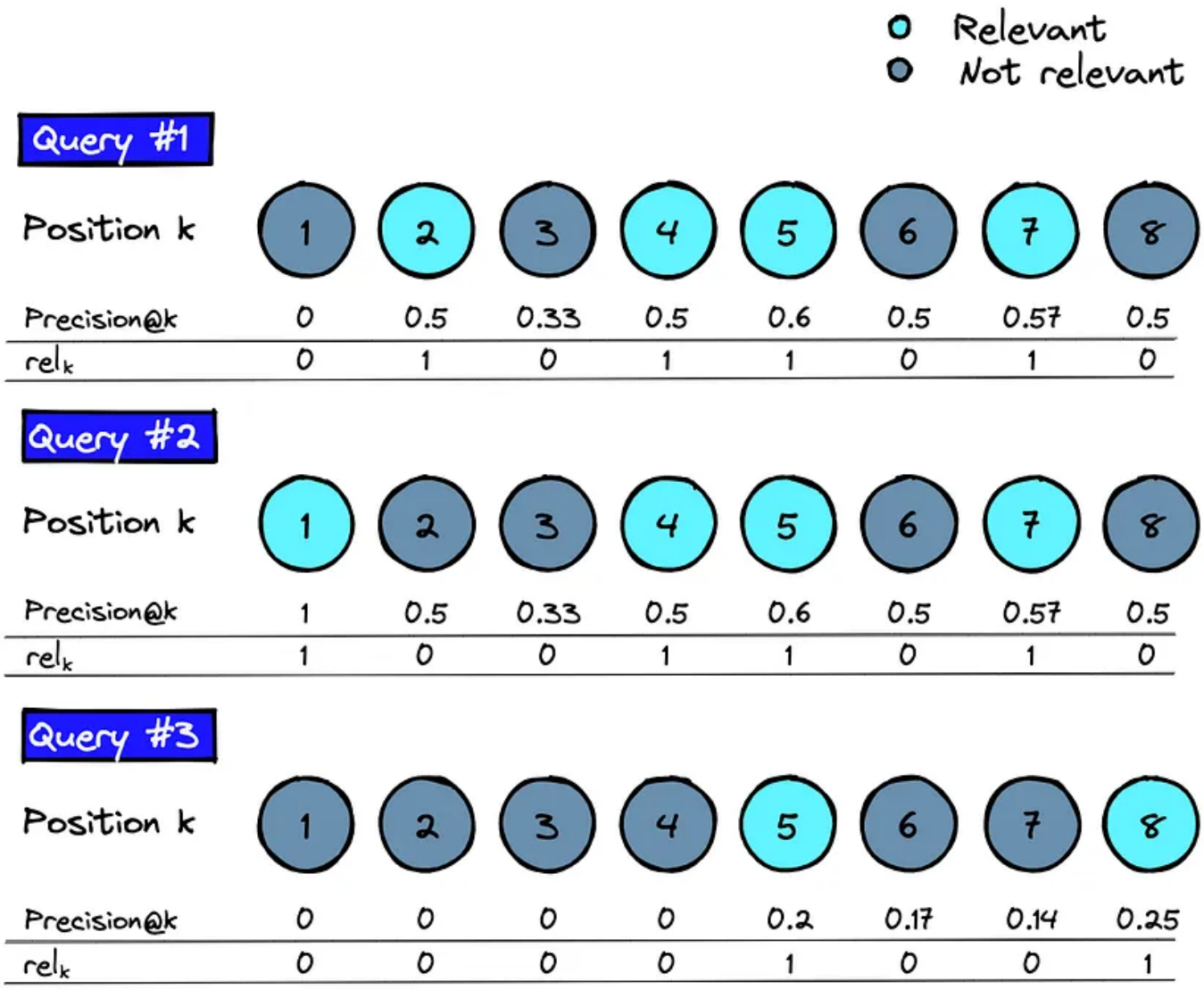
$$Precision@2 = \frac{p\hat{p}}{p\hat{p} + n\hat{p}} = \frac{p\hat{p}}{K} = \frac{1}{2} = 0.5$$

Note that the denominator in *precision@K* always equals K. Now that we have the *precision@K* value, we move on to the next step of calculating the Average Precision@K (*AP@K*):

$$AP@K = \frac{\sum_{k=1}^{K}(Precision@k * rel_k)}{number\ of\ relevant\ results}$$

Note that for *AP@K* we are taking the average *precision@k* score for all values of *k*=[1,..., *K*]. Meaning that for AP@5 we calculate *precision@k* where *k*=[1,2,3,4,5].

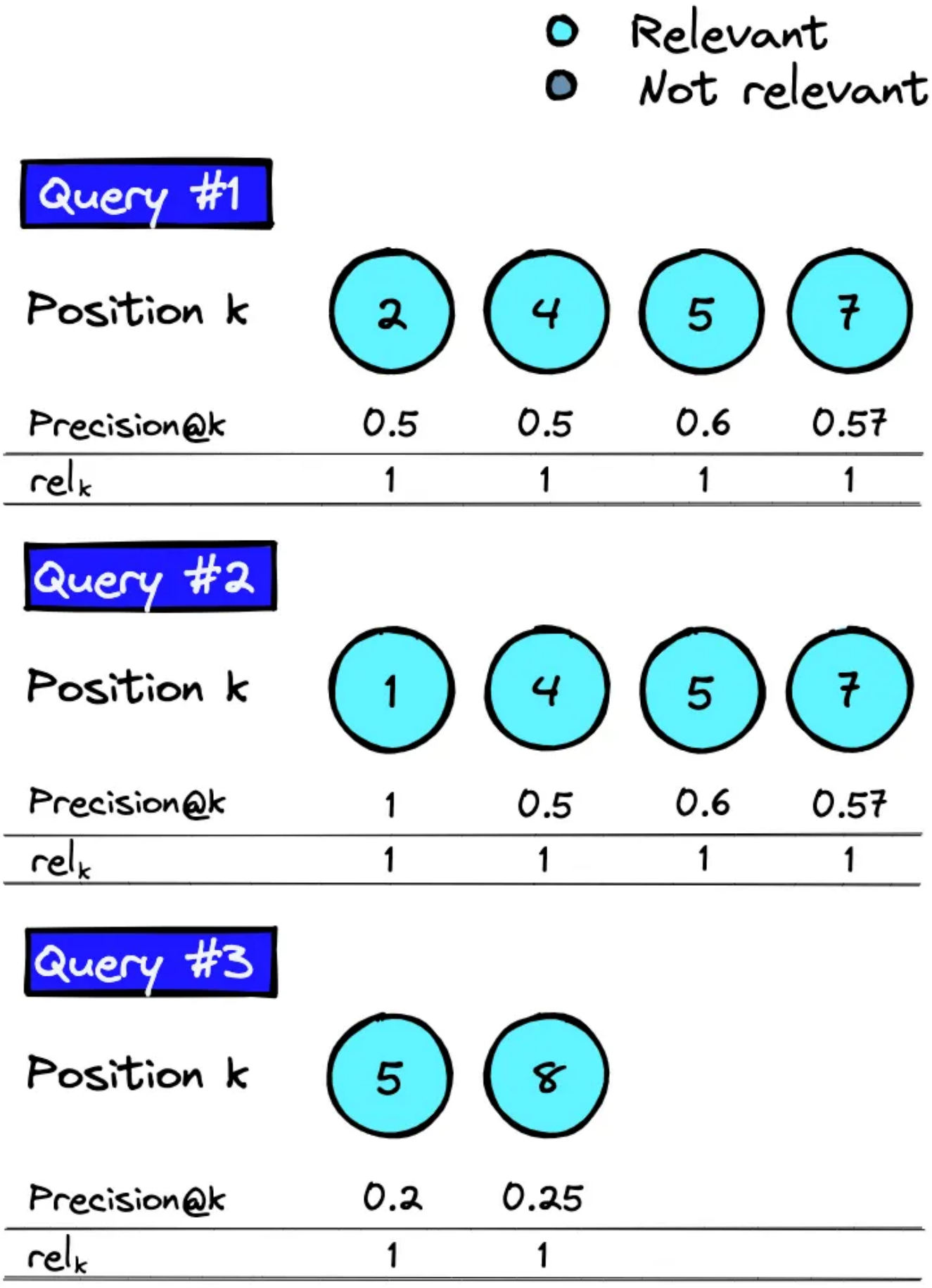
We have not seen the *rel-k* parameter before. It is a *relevance* parameter that (for *AP@K*) is equal to *1* when the *kth* item is relevant or *0* when it is not.



By Pinecone

We calculate precision@k and *rel-k* iteratively using *k*=[1,...,*K*].

As we multiply *precision@k* and *rel-k*, we only need to consider *precision@k* where the *kth* item is relevant.



Precision@k and rel_k values for all relevant items across three queries q = [1, ..., Q]

Given these values, for each query *q*, we can calculate the *AP@Kq* score where *K*=8 as :

$$AP@8_1 = \frac{0.5 * 1 + 0.5 * 1 + 0.6 * 1 + 0.57 * 1}{4} = 0.54$$

$$AP@8_2 = \frac{1 * 1 + 0.5 * 1 + 0.4 * 1 + 0.43 * 1}{4} = 0.67$$

$$AP@8_3 = \frac{0.2 * 1 + 0.25 * 1}{2} = 0.22$$

Each of these individual *AP@Kq* calculations produces a single **A**verage **P**recision@K score for each query q. To get the **M**ean **A**verage **P**recision@K (*MAP@K*) score for all queries, we simply divide by the number of queries *Q*:

$$MAP@K = \frac{1}{Q}\sum_{q=1}^{Q} AP@K_q = \frac{1}{3} * (0.54 + 0.67 + 0.22) = 0.48$$

That leaves us with a final *MAP@K* score of *0.48*. To calculate all of this with Python, we write:

```
# initialize variables
actual = [
    [2, 4, 5, 7],
    [1, 4, 5, 7],
    [5, 8]
]
Q = len(actual)
predicted = [1, 2, 3, 4, 5, 6, 7, 8]
k = 8
ap = []

# loop through and calculate AP for each query q
```

```
    for q in range(Q):
        ap_num = 0
        # loop through k values
        for x in range(k):
            # calculate precision@k
            act_set = set(actual[q])
            pred_set = set(predicted[:x+1])
            precision_at_k = len(act_set & pred_set) / (x+1)
            # calculate rel_k values
            if predicted[x] in actual[q]:
                rel_k = 1
            else:
                rel_k = 0
            # calculate numerator value for ap
            ap_num += precision_at_k * rel_k
        # now we calculate the AP value as the average of AP
        # numerator values
        ap_q = ap_num / len(actual[q])
        print(f"AP@{k}_{q+1} = {round(ap_q,2)}")
        ap.append(ap_q)
    # now we take the mean of all ap values to get mAP
    map_at_k = sum(ap) / Q
    # generate results
    print(f"mAP@{k} = {round(map_at_k, 2)}")
```

Output :

```
AP@8_1 = 0.54
AP@8_2 = 0.67
AP@8_3 = 0.23
mAP@8 = 0.48
```

Returning the same *MAP@K* score of 0.480.48.

Pros and Cons

MAP@K is a simple offline metric that allows us to consider the *order* of returned items. Making this ideal for use cases where we expect to return multiple relevant items.

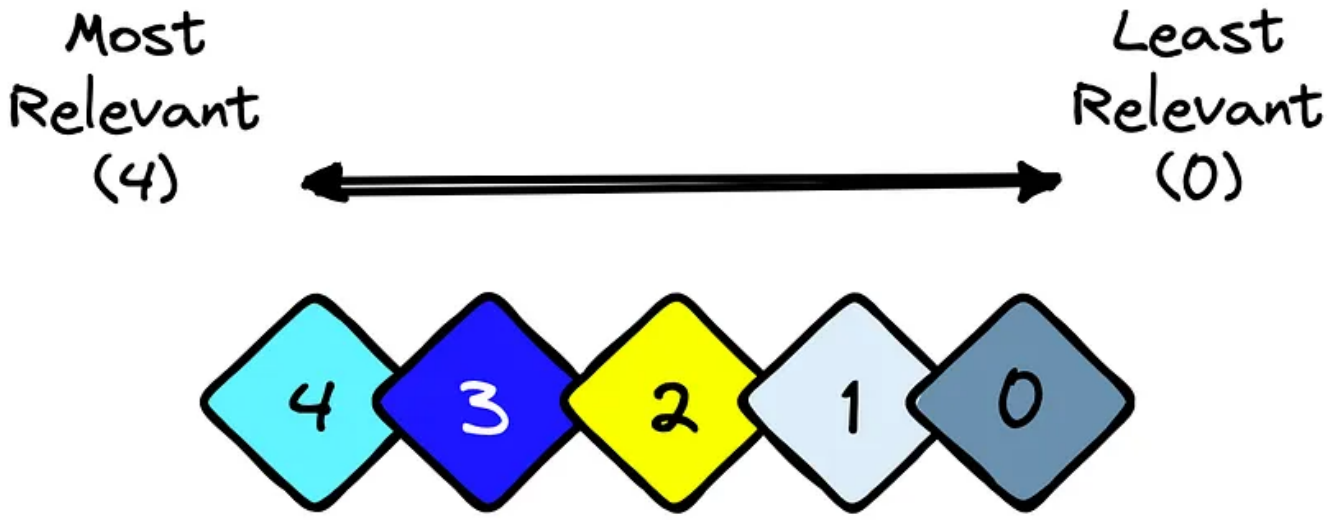
The primary disadvantage of MAP@K is the *rel-K* relevance parameter is binary. We must either view items as *relevant* or *irrelevant*. It does not allow for items to be slightly more/less relevant than others.

5.6. Normalized Discounted Cumulative Gain (NDCG@K)

Normalized Discounted Cumulative Gain @K (*NDCG@K*) is another *order-aware metric* that we can derive from a few simpler metrics. Starting with Cumulative Gain (*CG@K*) calculated like so:

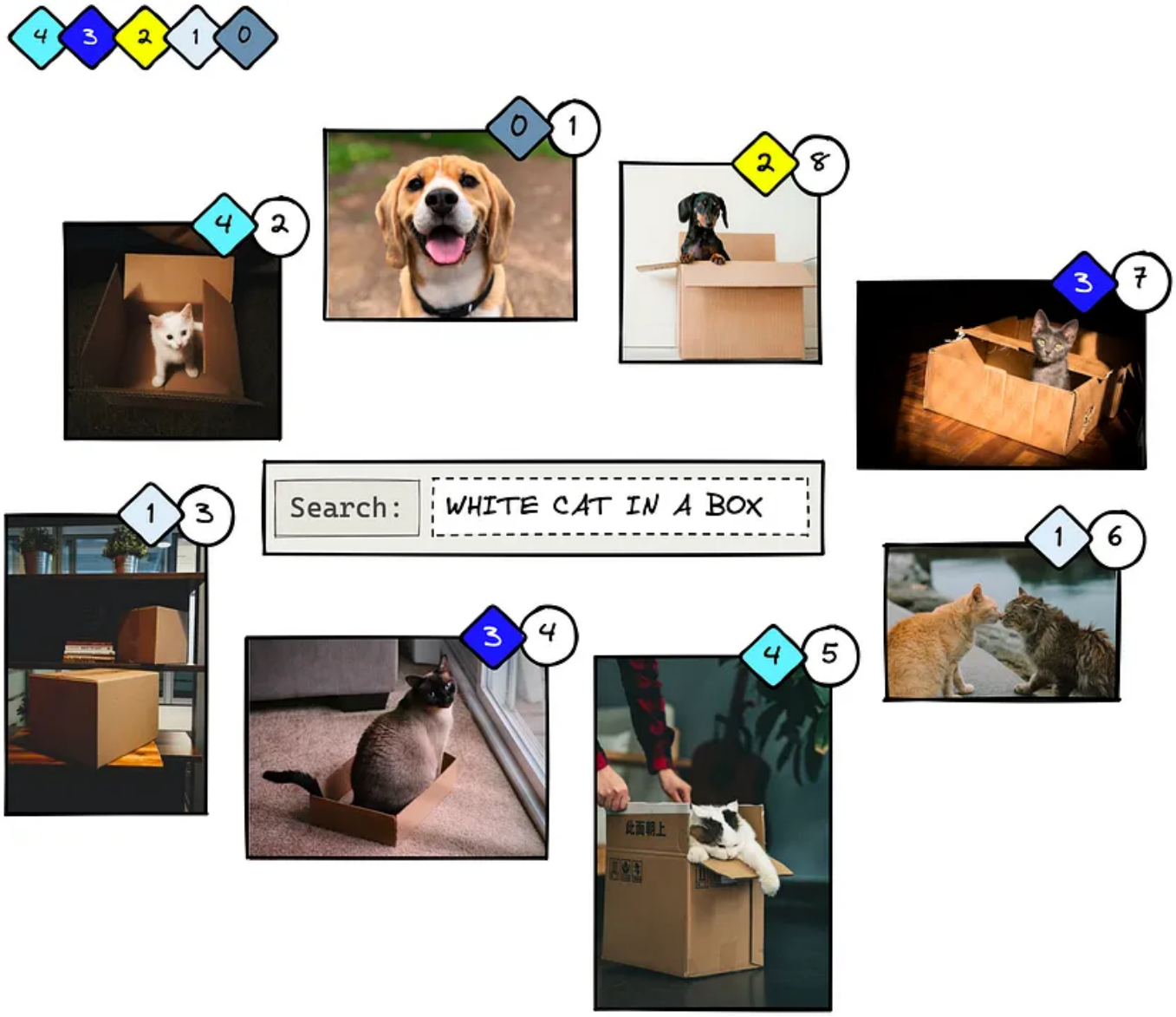
$$CG@K = \sum_{k=1}^{K} rel_k$$

The *rel-K* variable is different this time. It is a range of relevance ranks where *0* is the least relevant, and some higher value is the most relevant. The number of ranks does not matter; in our example, we use a range of 0→40→4.



Using rel_k we rank every item based on its relevance to a particular query q.

Let's try applying this to another example. We will use a similar eight-image dataset as before. The circled numbers represent the IR system's *predicted* ranking, and the diamond shapes represent the ****rel*k** *actual ranking*.

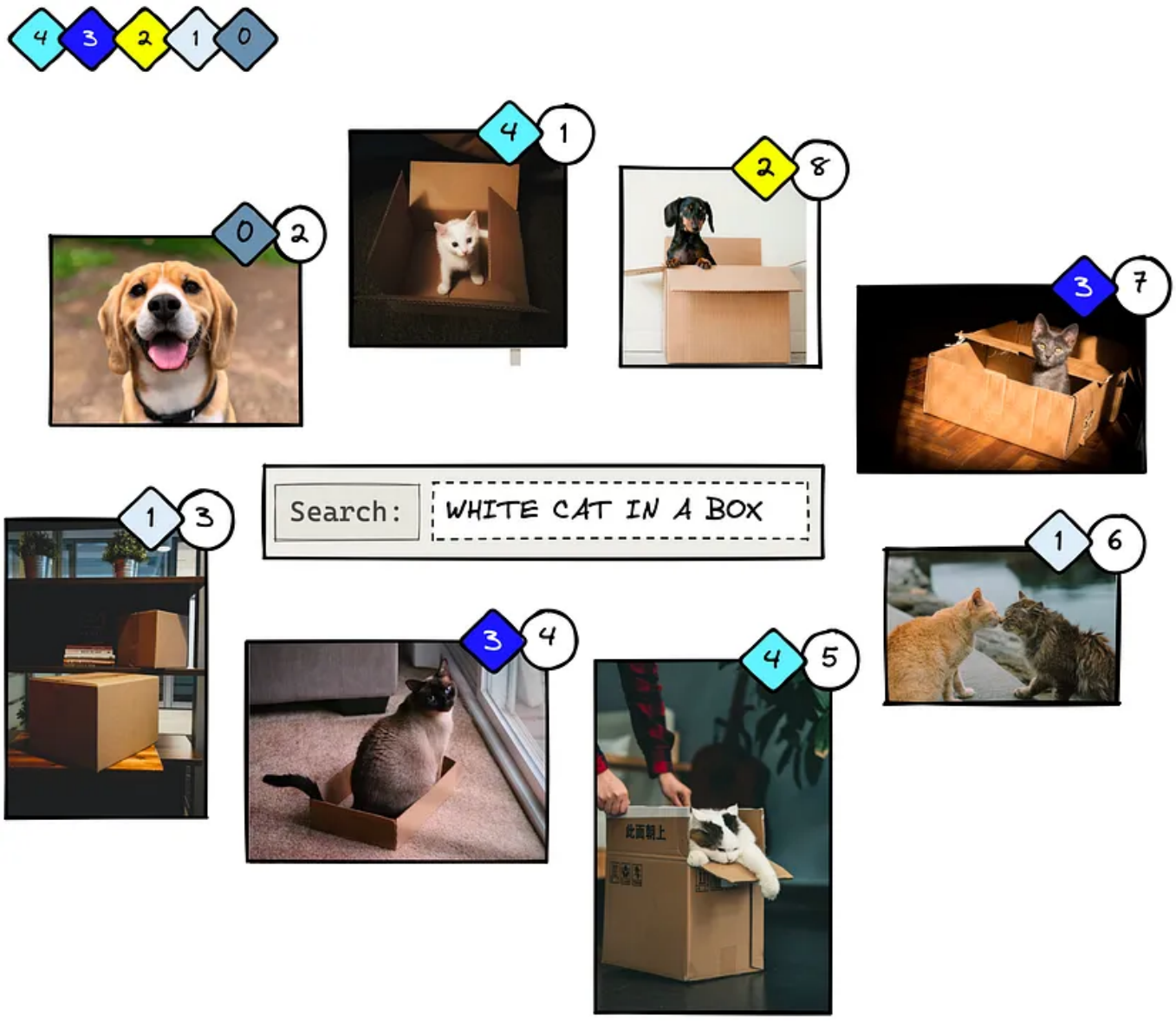


A small dataset with predicted ranks (circles) and actual ranks (diamonds).

To calculate the cumulative gain at position K ($CG@K$), we sum the relevance scores up to the *predicted* rank K. When K=2:

$$CG@2 = \sum_{k=1}^{2} rel_k = rel_1 + rel_2 = 0 + 4 = 4$$

It's important to note that *CG@K* is *not* order-aware. If we swap images *1* and *2*, we will return the same score when $K \geq 2$ despite having the more relevant item placed first.



Images 1 and 2 have been swapped.

$$CG@2 = \sum_{k=1}^{2} rel_k = rel_1 + rel_2 = 0 + 4 = 4$$

To handle this lack of order awareness, we modify the metric to create *DCG@K*, adding a penalty in the form of $log2(1+k)$ to the formula:

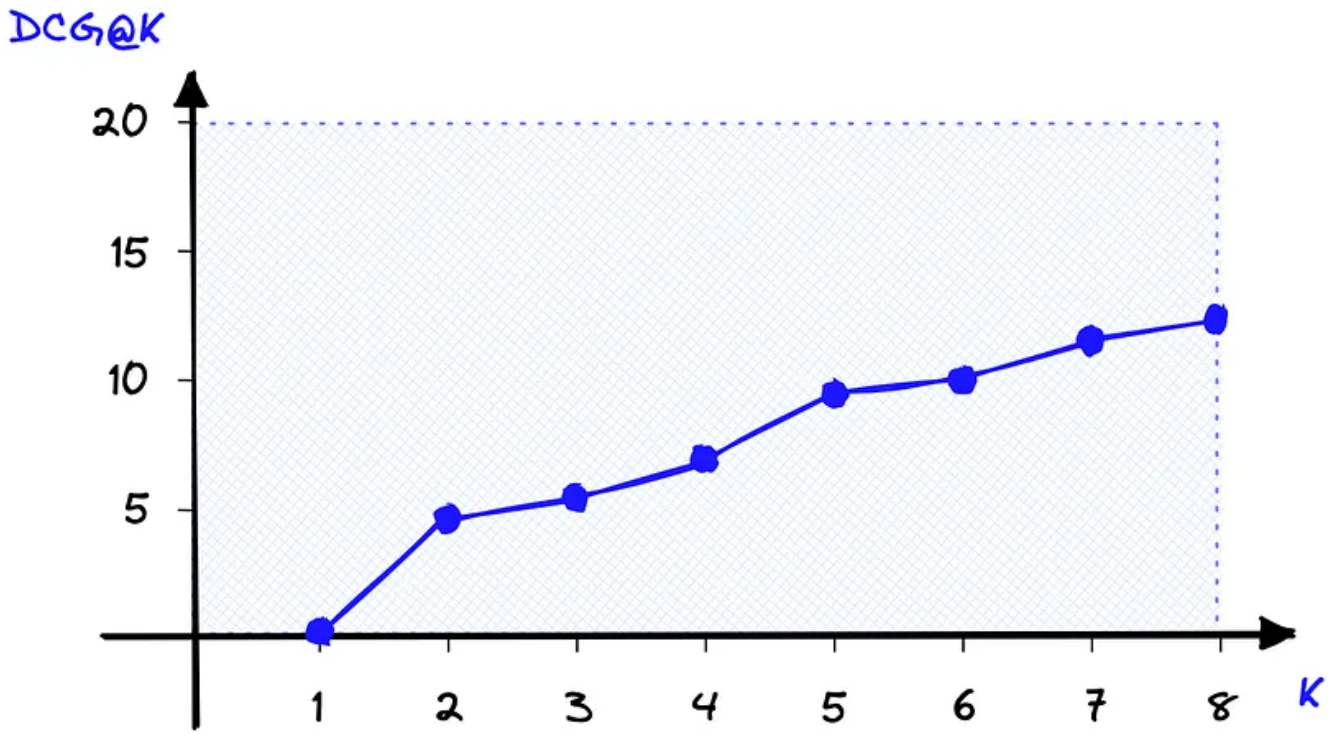
$$DCG@2 = \sum_{k=1}^{K} \frac{rel_k}{log_2(1+k)}$$

Now when we calculate *DCG@2* and swap the position of the first two images, we return different scores :

$$original: \ DCG@2 = \frac{0}{\log_2(1+1)} + \frac{4}{\log_2(1+2)} = 0 + 2.52 = 2.52$$

$$swapped: \ DCG@2 = \frac{4}{\log_2(1+1)} + \frac{0}{\log_2(1+2)} = 4 + 0 = 4$$

```
from math import log2
# initialize variables
relevance = [0, 7, 2, 4, 6, 1, 4, 3]
K = 8
dcg = 0
# loop through each item and calculate DCG
for k in range(1, K+1):
    rel_k = relevance[k-1]
    # calculate DCG
    dcg += rel_k / log2(1 + k)
```



DCG@K score as K increases using the query #1 order of results.

Using the *order-aware DCG@K* metric means the preferred swapped results returns a better score.

Unfortunately, *DCG@K* scores are very hard to interpret as their range depends on the variable *rel-k* range we chose for our data. We use the **N**ormalized **DCG@K** (*NDCG@K*) metric to fix this.

NDCG@K is a special modification of standard NDCG that cuts off any results whose rank is greater than K. This modification is prevalent in use-cases measuring search performance.

NDCG@K normalizes *DCG@K* using the **I**deal **DCG@K** (*IDCG@K*) rankings. For *IDCG@K*, we assume that the most relevant items are ranked highest and in order of relevance.

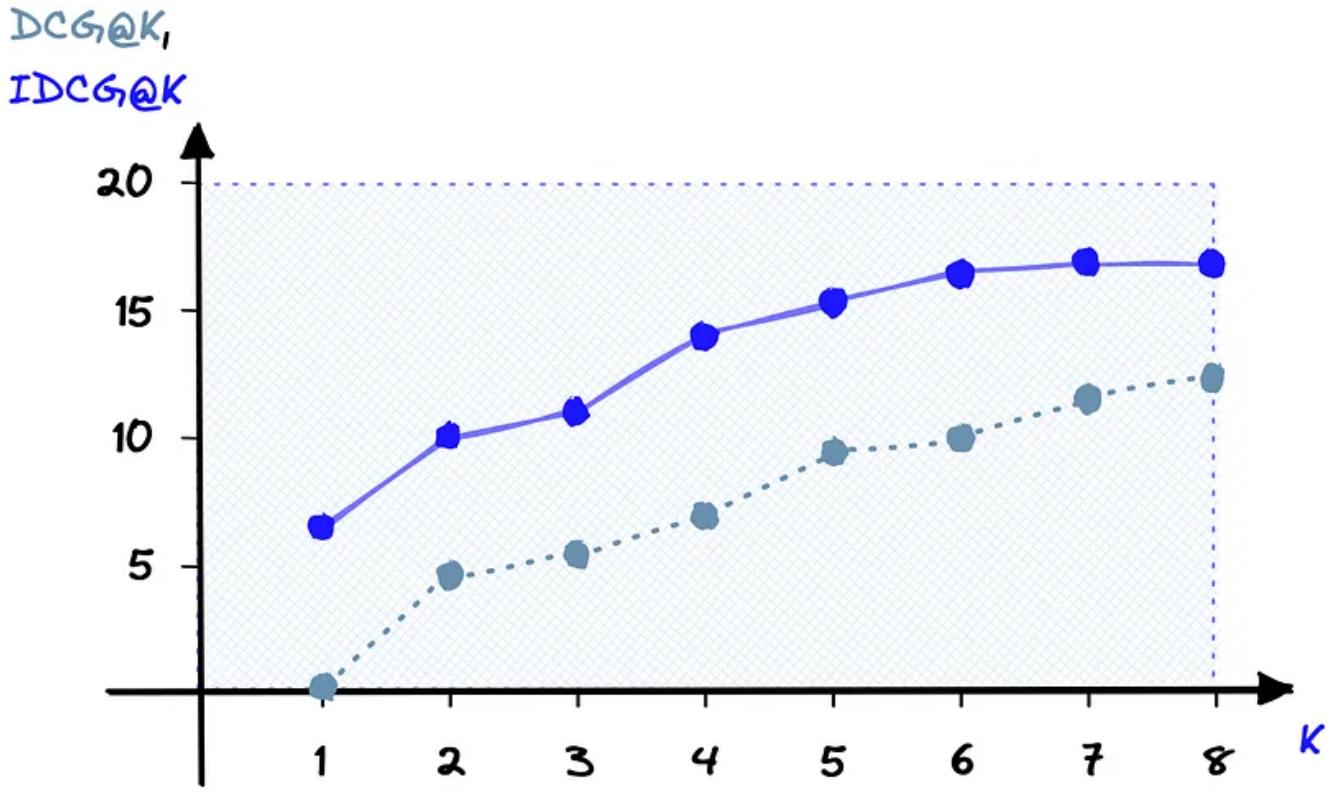
Calculating *IDCG@K* takes nothing more than reordering the assigned ranks and performing the same *DCG@K* calculation:

$$IDCG@2 = \frac{4}{\log_2(1+1)} + \frac{4}{\log_2(1+2)} = 4 + 2.52 = 6.52$$

```
# sort items in 'relevance' from most relevant to less relevant
ideal_relevance = sorted(relevance, reverse=True)

print(ideal_relevance)

idcg = 0
# as before, loop through each item and calculate *Ideal* DCG
for k in range(1, K+1):
    rel_k = ideal_relevance[k-1]
    # calculate DCG
    idcg += rel_k / log2(1 + k)
```



IDCG@K score as K increases compared against the DCG@K score calculated with using the query #1 order of results.

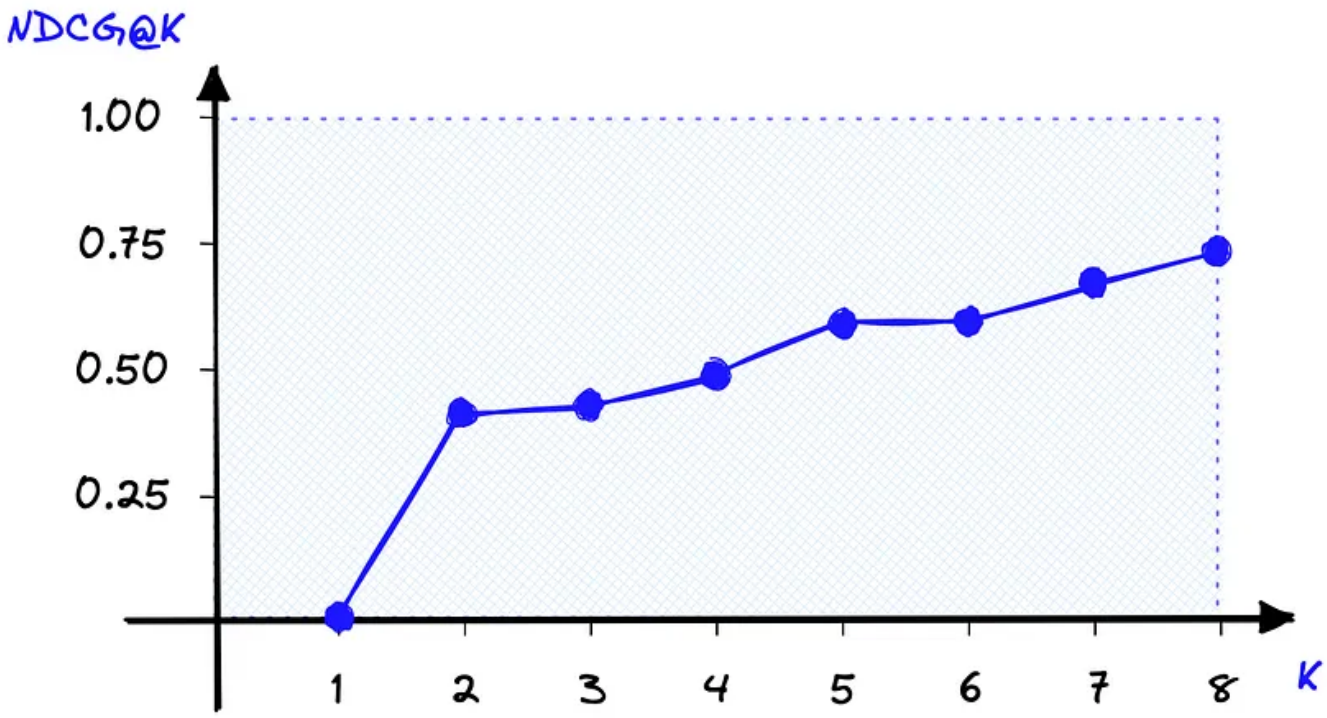
Now all we need to calculate *NDCG@K* is to normalize our *DCG@K* score using the *IDCG@K* score:

$$NDCG@K = \frac{DCG@K}{IDCG@K} = \frac{2.52}{6.52} = 0.39$$

```
dcg = 0
idcg = 0

for k in range(1, K+1):
    # calculate rel_k values
    rel_k = relevance[k-1]
    ideal_rel_k = ideal_relevance[k-1]
    # calculate dcg and idcg
    dcg += rel_k / log2(1 + k)
```

```
        idcg += ideal_rel_k / log2(1 + k)
        # calcualte ndcg
        ndcg = dcg / idcg
```



NDCG@K score as K increases is calculated by normalizing DCG@K using IDCG@K.

Using *NDCG@K*, we get a more interpretable result of *0.41*, where we know that *1.0* is the *best* score we can get with all items ranked perfectly (e.g., the *IDCG@K*).

Pros and Cons

NDCG@K is one of the most popular offline metrics for evaluating IR systems, in particular web search engines. That is because *NDCG@K* optimizes for highly relevant documents, is *order-aware*, and is easily interpretable.

However, there is a significant disadvantage to *NDCG@K*. Not only do we need to know which items are relevant for a particular query, but we need to know whether each item is more/less relevant than other items; the data requirements are more complex.

sentence 1	sentence 2	relevance label (simple)	relevance label (for NDCG)
white cat sitting in a box	a grey cat in a box	1	3
a box with a white cat sat inside	white cat sitting in a box	1	4
boxes stacked on top of eachother	cats fighting	0	0
two cats fighting	a white cat sat on the floor	0	1

Example of data for the other metrics (left) and the more complex data required for NDCG@K (right).

These are some of the most popular offline metrics for evaluating information retrieval systems. A single metric can be a good indicator of system performance. For even more confidence in retrieval performance, we can use several metrics, just as Spotify did with recall@1, recall@30, and MRR@30.

These metrics are still best supported with online metrics during A/B testing, which acts as *the next step* before deploying our retrieval system to the world. However, these offline metrics are the foundation behind any retrieval project.

Whether we're prototyping our very first product, or evaluating the latest iteration of Google search, evaluating our retrieval system with these metrics will help us deploy the best retrieval system possible.

Conclusion

In this blog we explored the search process in the Retrieval Augmented Generation (RAG) application, emphasizing semantic search using vector databases. It highlights advantages such as reduced processing time and real-time updates. Challenges include subpar responses to unique queries. Strategies to prevent issues involve monitoring query density and user feedback. Optimization efforts should focus on the building and post-production phases, including expanding the knowledge base. The blog also discusses types of search, semantic search, and a retrieve & re-rank pipeline. Pre-trained models and offline evaluation metrics like Recall@K are recommended. Overall, it emphasizes the need for continuous optimization and thorough evaluation in RAG applications.

Credits

In this blog post, we have compiled information from various sources, including research papers, technical blogs, official documentations, YouTube videos, and more. Each source has been appropriately credited beneath the corresponding images, with source links provided.

Below is a consolidated list of references:

1. https://dev.to/sfoteini/image-vector-similarity-search-with-azure-computer-vision-and-postgresql-12f7
2. https://www.pinecone.io/learn/retrieval-augmented-generation/
3. https://arize.com/blog-course/introduction-to-retrieval-augmented-generation/
4. https://thedataquarry.com/posts/vector-db-2/
5. https://www.sbert.net/examples/applications/retrieve_rerank/README.html

6. https://levelup.gitconnected.com/3-query-expansion-methods-implemented-using-langchain-to-improve-your-rag-81078c1330cd
7. https://www.pinecone.io/learn/offline-evaluation/

Thank you for reading!

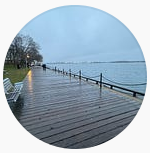
If this guide has enhanced your understanding of Python and Machine Learning:

- Please show your support with a clap 👏 or several claps!
- Your claps help me create more valuable content for our vibrant Python or ML community.
- Feel free to share this guide with fellow Python or AI / ML enthusiasts.
- Your feedback is invaluable — it inspires and guides my future posts.

Connect with me!

Vipra

Retrieval Augmented | Large Language Models | Search | Evaluation | Vector Database

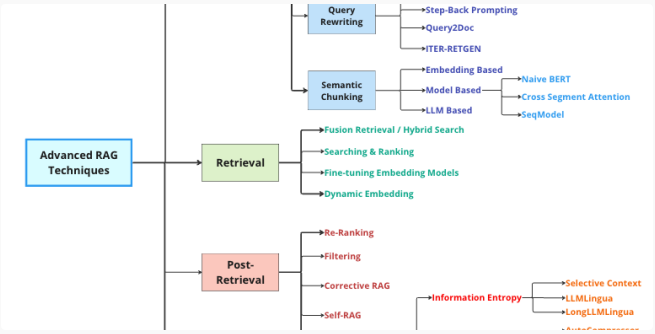


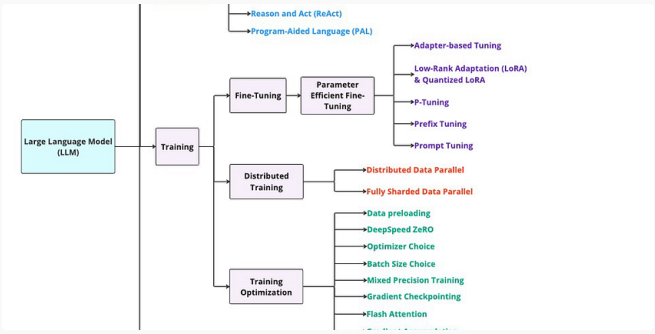
Written by Vipra Singh

1K Followers

Follow

More from Vipra Singh





Vipra Singh

Building LLM Applications: Advanced RAG (Part 10)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

· 48 min read · Apr 27, 2024

384 2

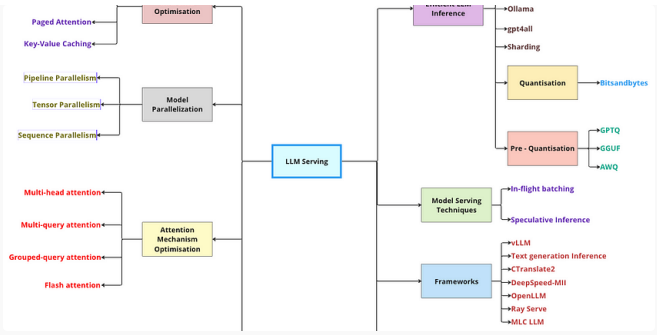
Vipra Singh

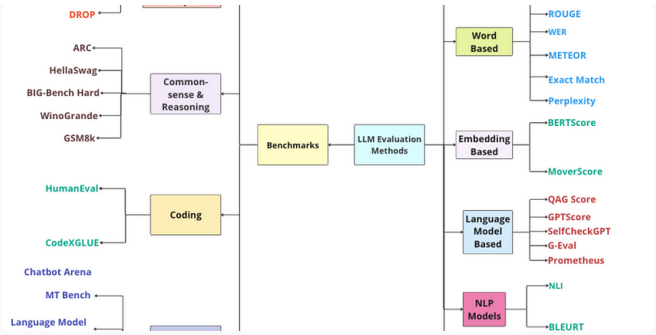
Building LLM Applications: Large Language Models (Part 6)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

38 min read · Feb 10, 2024

390 1





Vipra Singh

Building LLM Applications: Serving LLMs (Part 9)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

· 50 min read · Apr 17, 2024

546 3

Vipra Singh

Building LLM Applications: Evaluation (Part 8)

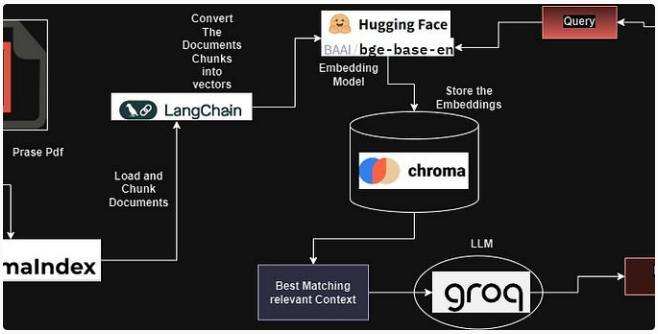
Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

· 48 min read · Apr 7, 2024

276 1

See all from Vipra Singh

Recommended from Medium



Plaban Nayak in The AI Forum

RAG on Complex PDF using LlamaParse, Langchain and Groq

Retrieval-Augmented Generation (RAG) is a new approach that leverages Large Languag...

13 min read · Apr 7, 2024

678 9

Paul Iusztin in Decoding ML

The 4 Advanced RAG Algorithms You Must Know to Implement

Implement from scratch 4 advanced RAG methods to optimize your retrieval and post-...

15 min read · May 4, 2024

1.4K 10

Lists

Natural Language Processing

1494 stories · 1011 saves

AI Regulation

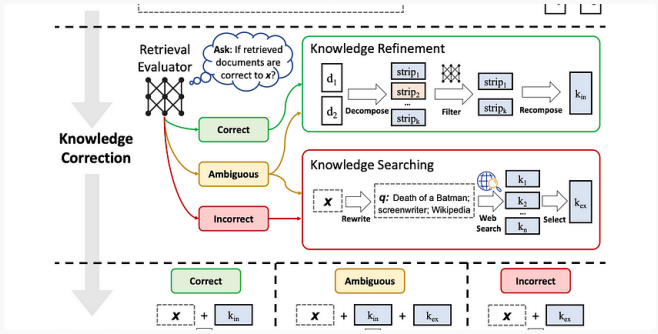
6 stories · 473 saves

ChatGPT prompts

47 stories · 1642 saves

Generative AI Recommended Reading

52 stories · 1106 saves



Barsha Rani Swain in GoPenAI

Advanced RAG: Corrective Retrieval Augmented Generation...

CRAG enhances the traditional RAG by introducing a retrieval evaluator to assess th...

10 min read · Apr 23, 2024

Tejpal Kumawat

Retrieval-Augmented Generation (RAG) from basics to advanced

Introduction:

12 min read · Feb 14, 2024

320 169

Ian Kelk

RAG Detective: Retrieval Augmented Generation with...

This article was produced as part of the final project for Harvard's AC215 Fall 2023 course.

16 min read · Dec 10, 2023

207 2

IVAN ILIN in Towards AI

Advanced RAG Techniques: an Illustrated Overview

A comprehensive study of the advanced retrieval augmented generation techniques...

19 min read · Dec 17, 2023

5.5K 37

See more recommendations