## LLM Settings

Below are the common settings you will come across when using different LLM providers:

**Temperature** - In short, the lower the temperature, the more deterministic the results in the sense that the highest probable next token is always picked. Increasing temperature could lead to more randomness, which encourages more diverse or creative outputs. You are essentially increasing the weights of the other possible tokens. In terms of application, you might want to use a lower temperature value for tasks like fact-based QA to encourage more factual and concise responses. For poem generation or other creative tasks, it might be beneficial to increase the temperature value.

**Top P** - A sampling technique with temperature, called nucleus sampling, where you can control how deterministic the model is. If you are looking for exact and factual answers keep this low. If you are looking for more diverse responses, increase to a higher value. If you use Top P it means that only the tokens comprising the top\_p probability mass are considered for responses, so a low top\_p value selects the most confident responses. This means that a high top\_p value will enable the model to look at more possible words, including less likely ones, leading to more diverse outputs.

The general recommendation is to alter temperature or Top P but not both.

**Max Length** - You can manage the number of tokens the model generates by adjusting the max length. Specifying a max length helps you prevent long or irrelevant responses and control costs.

**Stop Sequences** - A stop sequence is a string that stops the model from generating tokens. Specifying stop sequences is another way to control the length and structure of the model's response. For example, you can tell the model to generate lists that have no more than 10 items by adding "11" as a stop sequence.

**Frequency Penalty** - The frequency penalty applies a penalty on the next token proportional to how many times that token already appeared in the response and prompt. The higher the frequency penalty, the less likely a word will appear again. This setting reduces the repetition of words in the model's response by giving tokens that appear more a higher penalty.

**Presence Penalty** - The presence penalty also applies a penalty on repeated tokens but, unlike the frequency penalty, the penalty is the same for all repeated tokens. A token that appears twice and a token that appears 10 times are penalized the same. This setting prevents the model from repeating phrases too often in its response. If you want the model to generate diverse or creative text, you might want to use a higher presence penalty. Or, if you need the model to stay focused, try using a lower presence penalty.

Similar to temperature and top\_p, the general recommendation is to alter the frequency or presence penalty but not both.

## Prompting an LLM

You can achieve a lot with simple prompts, but the quality of results depends on how much information you provide it and how well-crafted the prompt is. A prompt can contain information like the *instruction* or *question* you are passing to the model and include other details such as *context*, *inputs*, or *examples*. You can use these elements to instruct the model more effectively to improve the quality of results.

Let's get started by going over a basic example of a simple prompt:

*Prompt*

The sky is

*Output:*

blue.

## Zero-Shot Prompting

Large language models (LLMs) today, such as GPT-3.5 Turbo, GPT-4, and Claude 3, are tuned to follow instructions and are trained on large amounts of data. Large-scale training makes these models capable of performing some tasks in a "zero-shot" manner. Zero-shot prompting means that the prompt used to interact with the model won't contain examples or demonstrations. The zero-shot prompt directly instructs the model to perform a task without any additional examples to steer it.

We tried a few zero-shot examples in the previous section. Here is one of the examples (ie., text classification) we used:

*Prompt:*

*Classify the text into neutral, negative or positive.*

*Text: I think the vacation is okay.*

*Sentiment:*

*Output:*

*Neutral*

Note that in the prompt above we didn't provide the model with any examples of text alongside their classifications, the LLM already understands "sentiment" -- that's the zero-shot capabilities at work.

## Few-Shot Prompting

Let's demonstrate few-shot prompting via an example that was presented in [Brown et al. 2020](https://arxiv.org/abs/2005.14165). In the example, the task is to correctly use a new word in a sentence.

*Prompt:*

*A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is:*

*We were traveling in Africa and we saw these very cute whatpus.*

*To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:*

*Output:*

*When we won the game, we all started to farduddle in celebration.*

We can observe that the model has somehow learned how to perform the task by providing it with just one example (i.e., 1-shot). For more difficult tasks, we can experiment with increasing the demonstrations (e.g., 3-shot, 5-shot, 10-shot, etc.).

Following the findings from [Min et al. (2022)](https://arxiv.org/abs/2202.12837), here are a few more tips about demonstrations/exemplars when doing few-shot:

* "the label space and the distribution of the input text specified by the demonstrations are both important (regardless of whether the labels are correct for individual inputs)"
* the format you use also plays a key role in performance, even if you just use random labels, this is much better than no labels at all.
* additional results show that selecting random labels from a true distribution of labels (instead of a uniform distribution) also helps.

## Chain-of-Thought Prompting

Introduced in [Wei et al. (2022)](https://arxiv.org/abs/2201.11903), chain-of-thought (CoT) prompting enables complex reasoning capabilities through intermediate reasoning steps. You can combine it with few-shot prompting to get better results on more complex tasks that require reasoning before responding.

*Prompt:*

*The odd numbers in this group add up to an even number: 4, 8, 9, 15, 12, 2, 1.*

*A: Adding all the odd numbers (9, 15, 1) gives 25. The answer is False.*

*The odd numbers in this group add up to an even number: 17, 10, 19, 4, 8, 12, 24.*

*A: Adding all the odd numbers (17, 19) gives 36. The answer is True.*

*The odd numbers in this group add up to an even number: 16, 11, 14, 4, 8, 13, 24.*

*A: Adding all the odd numbers (11, 13) gives 24. The answer is True.*

*The odd numbers in this group add up to an even number: 17, 9, 10, 12, 13, 4, 2.*

*A: Adding all the odd numbers (17, 9, 13) gives 39. The answer is False.*

*The odd numbers in this group add up to an even number: 15, 32, 5, 13, 82, 7, 1.*

*A:*

*Output:*

*Adding all the odd numbers (15, 5, 13, 7, 1) gives 41. The answer is False.*

One recent idea that came out more recently is the idea of [zero-shot CoT](https://arxiv.org/abs/2205.11916) (Kojima et al. 2022) that essentially involves adding "Let's think step by step" to the original prompt.

*Prompt:*

*I went to the market and bought 10 apples. I gave 2 apples to the neighbor and 2 to the repairman. I then went and bought 5 more apples and ate 1. How many apples did I remain with?*

*Let's think step by step.*

*Output:*

*First, you started with 10 apples.*

*You gave away 2 apples to the neighbor and 2 to the repairman, so you had 6 apples left.*

*Then you bought 5 more apples, so now you had 11 apples.*

*Finally, you ate 1 apple, so you would remain with 10 apples.*

**Automatic Chain-of-Thought (Auto-CoT)**

When applying chain-of-thought prompting with demonstrations, the process involves hand-crafting effective and diverse examples. This manual effort could lead to suboptimal solutions. [Zhang et al. (2022)](https://arxiv.org/abs/2210.03493) propose an approach to eliminate manual efforts by leveraging LLMs with "Let's think step by step" prompt to generate reasoning chains for demonstrations one by one. This automatic process can still end up with mistakes in generated chains. To mitigate the effects of the mistakes, the diversity of demonstrations matter. This work proposes Auto-CoT, which samples questions with diversity and generates reasoning chains to construct the demonstrations.

Auto-CoT consists of two main stages:

* Stage 1): **question clustering**: partition questions of a given dataset into a few clusters
* Stage 2): **demonstration sampling**: select a representative question from each cluster and generate its reasoning chain using Zero-Shot-CoT with simple heuristics

The simple heuristics could be length of questions (e.g., 60 tokens) and number of steps in rationale (e.g., 5 reasoning steps). This encourages the model to use simple and accurate demonstrations.

## What is RAG?

LLMs, although capable of generating text that is both meaningful and grammatically correct, these LLMs suffer from a problem called hallucination. Hallucination in LLMs is the concept where the LLMs confidently generate wrong answers, that is they make up wrong answers in a way that makes us believe that it is true. This has been a major problem since the introduction of the LLMs. These hallucinations lead to incorrect and factually wrong answers. Hence Retrieval Augmented Generation was introduced.

In RAG, we take a list of documents/chunks of documents and encode these textual documents into a numerical representation called vector embeddings, where a single vector embedding represents a single chunk of document and stores them in a database called vector store. The models required for encoding these chunks into embeddings are called encoding models or bi-encoders. These encoders are trained on a large corpus of data, thus making them powerful enough to encode the chunks of documents in a single vector embedding representation.

## Explain Semantic Chunking

In order to abide by the context window of the LLM , we usually break text into smaller parts / pieces which is called chunking.

Different chunking methods:

* Fixed size chunking
* Recursive Chunking
* Document Specific Chunking
* Semantic Chunking
* Agentic Chunking

Semantic chunking involves taking the embeddings of every sentence in the document, comparing the similarity of all sentences with each other, and then grouping sentences with the most similar embeddings together. By focusing on the text’s meaning and context, Semantic Chunking significantly enhances the quality of retrieval. It’s a top-notch choice when maintaining the semantic integrity of the text is vital.

The hypothesis here is we can use embeddings of individual sentences to make more meaningful chunks. Basic idea is as follows :-

1. *Split the documents into sentences based on separators(.,?,!)*
2. *Index each sentence based on position.*
3. *Group: Choose how many sentences to be on either side. Add a buffer of sentences on either side of our selected sentence.*
4. *Calculate distance between group of sentences.*
5. *Merge groups based on similarity i.e. keep similar sentences together.*
6. *Split the sentences that are not similar.*

## Keyword-Based Retrieval

This retrieval type uses a keyword-based retriever, also known as a sparse retriever. An example of such a retriever is [BM25Retriever](https://docs.cloud.deepset.ai/docs/retriever#keyword-based-retrievers).

Sparse retrievers work with keywords, looking for words shared between the document and the query. They operate on a bag-of-words level and don’t consider the order of words or their contextual meanings, which means they may not capture semantic nuances as effectively as dense retrievers.

These retrievers don’t need any training and are fast and effective. They can work on any language and any domain.

## Vector-Based Retrieval

This retrieval type relies on vector-based, or dense, retrievers, such as [EmbeddingRetriever](https://docs.cloud.deepset.ai/docs/retriever#vector-based-retrievers). Dense retrievers use a model to transform both the documents and the query into numerical vectors (embeddings). Then, they compare both embeddings and, based on that, fetch the documents most similar to the query.

Dense retrievers are very good at capturing nuances in queries and documents, recognizing similarities that go beyond keyword matching. They can recognize contextual and semantic information about words and their relationships within a sentence.

Unlike sparse retrievers, dense retrievers need to be trained. This means they perform best on the domain and language they were trained on. They’re also more computationally expensive than keyword-based retrievers.

## Hybrid Retrieval

Sparse retrievers are fast and can quickly reduce the number of candidate documents. Dense retrievers are better at capturing semantic nuances, thus improving the relevance of search results.

For example, when searching for product IDs, keyword search is best. When given the query “P12642”, a sparse retriever would fetch “Miura climbing shoes” as a result. Dense retrievers would be thrown off by such a query since they can return results with a similar product ID.

On the other hand, a query like “What are EVs?” would be easier for vector-based retrievers. They would retrieve results like “Electric cars are..”, while sparse retrievers would look for the exact keyword match.

Combining both retrieval methods in one system makes it more robust to different kinds of queries and documents.

Once the retrievers fetch the most relevant documents, you can use a combination strategy to produce the final ranking and return the top documents as search results.

A good use case for hybrid retrieval is when your documents are from a niche domain, and it’s unlikely the model was trained on it. Hybrid retrieval saves you the time and money you’d need to train or fine-tune a model and it’s a good trade-off between speed and accuracy.

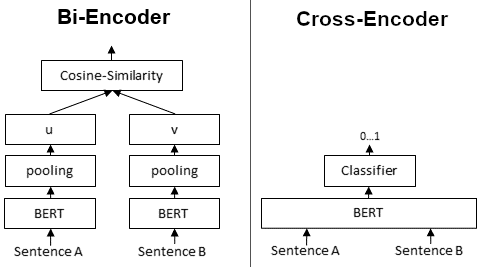
## Cross-Encoders

Cross-Encoders are used for sentence pair scoring and sentence pair classification tasks.

**Bi-Encoder vs. Cross-Encoder**

First, it is important to understand the difference between Bi- and Cross-Encoder.

**Bi-Encoders** produce for a given sentence a sentence embedding. We pass to a BERT independently the sentences A and B, which result in the sentence embeddings u and v. These sentence embedding can then be compared using cosine similarity:



In contrast, for a **Cross-Encoder**, we pass both sentences simultaneously to the Transformer network. It produces then an output value between 0 and 1 indicating the similarity of the input sentence pair:

A **Cross-Encoder does not produce a sentence embedding**. Also, we are not able to pass individual sentences to a Cross-Encoder.

As detailed in our [paper](https://arxiv.org/abs/1908.10084), Cross-Encoder achieve better performances than Bi-Encoders. However, for many application they are not practical as they do not produce embeddings we could e.g. index or efficiently compare using cosine similarity.

**When to use Cross- / Bi-Encoders?**

Cross-Encoders can be used whenever you have a pre-defined set of sentence pairs you want to score. For example, you have 100 sentence pairs and you want to get similarity scores for these 100 pairs.

Bi-Encoders are used whenever you need a sentence embedding in a vector space for efficient comparison. Applications are for example Information Retrieval / Semantic Search or Clustering. Cross-Encoders would be the wrong choice for these application: Clustering 10,000 sentence with CrossEncoders would require computing similarity scores for about 50 Million sentence combinations, which takes about 65 hours. With a Bi-Encoder, you compute the embedding for each sentence, which takes only 5 seconds. You can then perform the clustering.

## What is Reciprocal Rank Fusion?

Reciprocal Rank Fusion is a rank aggregation method that combines rankings from multiple sources into a single, unified ranking. In the context of RAG, these sources typically use different retrieval models or approaches.

**The RRF Formula**

The core of RRF is captured in its formula:

RRF(d) = Σ(r ∈ R) 1 / (k + r(d))

*Where:  
- d is a document  
- R is the set of rankers (retrievers)  
- k is a constant (typically 60)  
- r(d) is the rank of document d in ranker r*

**How RRF Works in RAG**

Let’s break down the process of using RRF in a RAG system:

1. **User Query**: The process begins when a user inputs a question or query.

2. **Multiple Retrievers**: The query is sent to multiple retrievers. These could be different retrieval models (e.g., dense, sparse, hybrid).

3. **Individual Rankings**: Each retriever produces its own ranking of relevant documents.

4. **RRF Fusion**: The rankings from all retrievers are combined using the RRF formula.

5. **Final Ranking**: A unified ranking is produced based on the RRF scores.

6. **Generation**: The generative model uses the top-ranked documents to produce the final answer.

**Mathematical Intuition Behind RRF**

Understanding the mathematical intuition behind RRF helps explain why it’s effective:

**1. Reciprocal Ranking**

Using 1/(rank + k), RRF gives more weight to higher ranks (lower rank numbers). This ensures that documents ranked highly by multiple retrievers are favoured in the final ranking.

**2. Diminishing Returns**

The contribution to the score decreases non-linearly as rank increases. This model shows the intuition that the difference in relevance between ranks 1 and 2 is likely larger than between ranks 100 and 101.

**3. Rank Aggregation**

By summing the reciprocal ranks across all retrievers, RRF effectively combines evidence from multiple sources. This makes the final ranking more robust and less susceptible to the quirks or biases of any single retriever.

**4. Normalization**

The constant k acts as a smoothing factor. It prevents any single retriever from dominating the results and helps handle ties more gracefully, especially among lower-ranked items.

**The Mystery of k = 60**

One aspect of RRF that often raises questions is the choice of k = 60. While this value isn’t set in stone, it’s commonly used due to several factors:

**1. Empirical Performance**

Studies have shown that k = 60 performs well across various datasets and retrieval tasks.

**2. Balancing Influence**

It provides a good balance between the influence of top-ranked and lower-ranked items. For example:  
- For rank 1: 1/(1+60) ≈ 0.0164  
- For rank 10: 1/(10+60) ≈ 0.0143  
- For rank 100: 1/(100+60) ≈ 0.00625

**3. Effective Tie-Breaking**

k = 60 helps break ties effectively, especially for lower-ranked items where small differences in the original rankings might not be significant.

**4. Robustness**

This value has shown to be robust across different types of retrieval systems and data distributions.

It’s worth noting that while k = 60 is common, the optimal value can vary depending on the specific application and data characteristics. Some systems may benefit from tuning this parameter.

## Metrics in Information 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.

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 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*.

## Recall@K

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

Recall@K=truePositives/(truePositives+falseNegatives)

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][1,...,N].

**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.

## 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=1/Q\*∑​1/rank\_q​

Q is the number of queries, q a specific query, and rank\_q​ the rank of the first \*actual relevant\* result for query q.

**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](https://www.pinecone.io/learn/series/nlp/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 recommendation 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.

## Mean Average Precision (MAP)

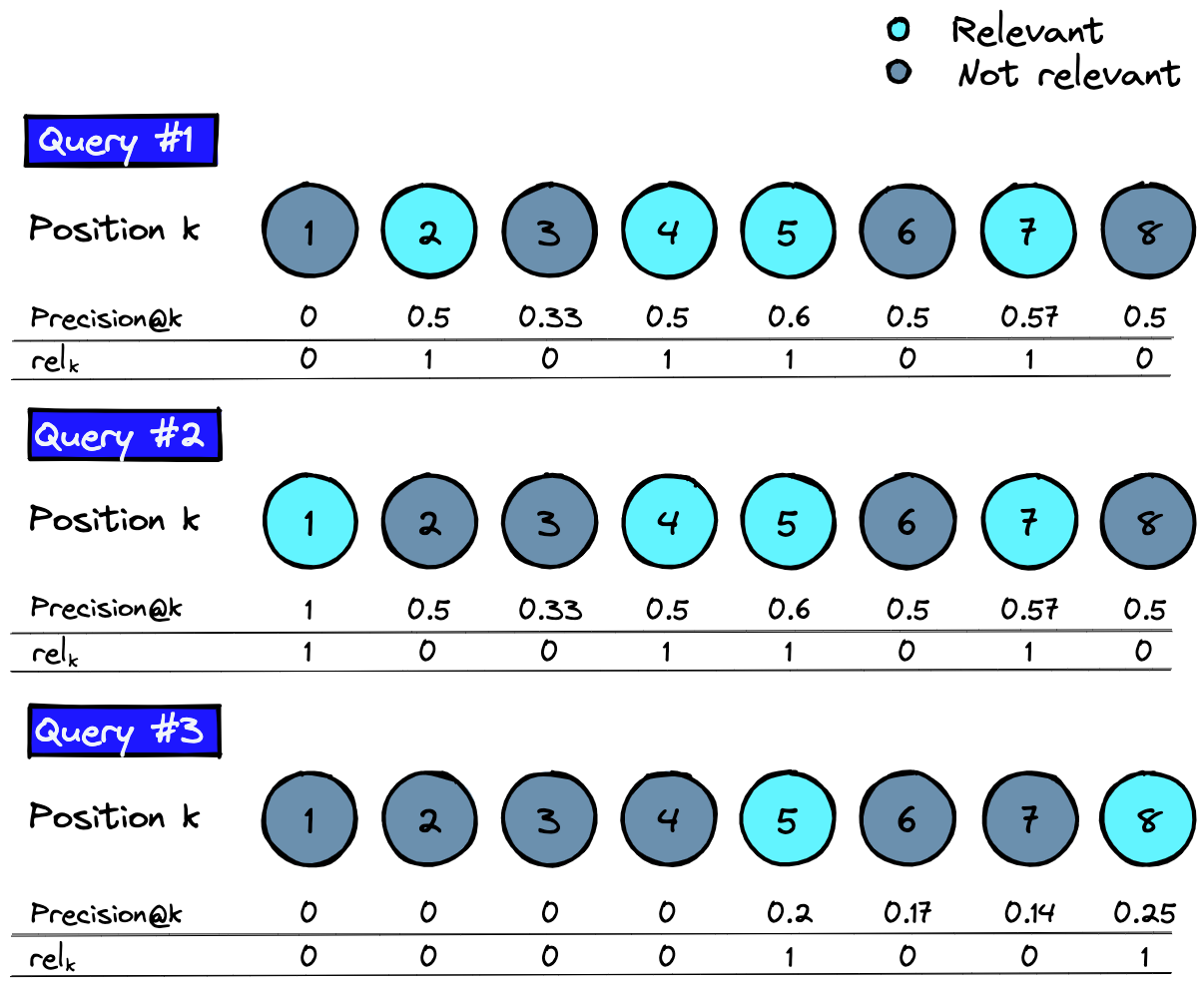
**M**ean **A**verage **P**recision@K (*MAP@K*) is another popular *order-aware* metric.

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

Precision@K=truePositives/(truePositives+falsePositives)

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

AP@K=∑(Precision@k∗relevance\_k)/(number of relevant results)



**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 relKrelK​ 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

## Normalized Discounted Cumulative Gain (NDCG@K)

***N****ormalized* ***D****iscounted* ***C****umulative* ***G****ain* ***@K*** (NDCG@KNDCG@K) is another *order-aware metric* that we can derive from a few simpler metrics. Starting with **C**umulative **G**ain (CG@KCG@K) calculated like so:

CG@K=∑relevance\_k

The relevance\_k​ variable 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→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=∑relevance\_k / log2(1+k)

Using the *order-aware* DCG@KDCG@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 relkrelk​ 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.

**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.

## Fine-tuning

Research shows that the pattern-recognition abilities of foundation language models are so powerful that they sometimes require relatively little additional training to learn specific tasks. That additional training helps the model make better predictions on a specific task. This additional training, called [**fine-tuning**](https://developers.google.com/machine-learning/glossary#fine-tuning), unlocks an LLM's practical side.

Fine-tuning trains on examples *specific* to the task your application will perform. Engineers can sometimes fine-tune a foundation LLM on just a few hundred or a few thousand training examples.

Despite the relatively tiny number of training examples, standard fine-tuning is often computationally expensive. That's because standard fine-tuning involves updating the weight and bias of every parameter on each [**backpropagation**](https://developers.google.com/machine-learning/glossary#backpropagation) iteration. Fortunately, a smarter process called [**parameter-efficient tuning**](https://developers.google.com/machine-learning/glossary#parameter-efficient-tuning) can fine-tune an LLM by adjusting only a *subset* of parameters on each backpropagation iteration.

A fine-tuned model's predictions are usually better than the foundation LLM's predictions. However, a fine-tuned model contains the same number of parameters as the foundation LLM. So, if a foundation LLM contains ten billion parameters, then the fine-tuned version will also contain ten billion parameters.

## Distillation

Most fine-tuned LLMs contain enormous numbers of parameters. Consequently, foundation LLMs require enormous computational and environmental resources to generate predictions. Note that large swaths of those parameters are typically irrelevant for a specific application.

[**Distillation**](https://developers.google.com/machine-learning/glossary#distillation) creates a smaller version of an LLM. The distilled LLM generates predictions much faster and requires fewer computational and environmental resources than the full LLM. However, the distilled model's predictions are generally not quite as good as the original LLM's predictions. Recall that LLMs with more parameters almost always generate better predictions than LLMs with fewer parameters.

The most common form of distillation uses bulk inference to label data. This labeled data is then used to train a new, smaller model (known as the student model) that can be more affordably served. The labeled data serves as a channel by which the larger model (known as the teacher model) funnels its knowledge to the smaller model.

For example, suppose you need an online toxicity scorer for automatic moderation of comments. In this case, you can use a large offline toxicity scorer to label training data. Then, you can use that training data to distill a toxicity scorer model small enough to be served and handle live traffic.

A teacher model can sometimes provide more labeled data than it was trained on. Alternatively, a teacher model can funnel a numerical score instead of a binary label to the student model. A numerical score provides a richer training signal than a binary label, enabling the student model to predict not only positive and negative classes but also borderline classes.

## Offline inference

The number of parameters in an LLM is sometimes so large that [**online inference**](https://developers.google.com/machine-learning/glossary#online-inference) is too slow to be practical for real-world tasks like regression or classification. Consequently, many engineering teams rely on [**offline inference**](https://developers.google.com/machine-learning/glossary#offline-inference) (also known as *bulk inference* or *static inference*) instead. In other words, rather than responding to queries at serving time, the trained model makes predictions in advance and then caches those predictions.

It doesn't matter if it takes a long time for an LLM to complete its task if the LLM only has to perform the task once a week or once a month.

For example, Google Search [used an LLM](https://blog.google/products/search/how-mum-improved-google-searches-vaccine-information/) to perform offline inference in order to cache a list of over 800 synonyms for Covid vaccines in more than 50 languages. Google Search then used the cached list to identify queries about vaccines in live traffic.

## Distributed Data Parallel

[DistributedDataParallel](https://pytorch.org/docs/stable/nn.html#module-torch.nn.parallel) (DDP) is a powerful module in PyTorch that allows you to parallelize your model across multiple machines, making it perfect for large-scale deep learning applications. To use DDP, you’ll need to spawn multiple processes and create a single instance of DDP per process.

But how does it work? DDP uses collective communications from the [torch.distributed](https://pytorch.org/tutorials/intermediate/dist_tuto.html) package to synchronize gradients and buffers across all processes. This means that each process will have its own copy of the model, but they’ll all work together to train the model as if it were on a single machine.

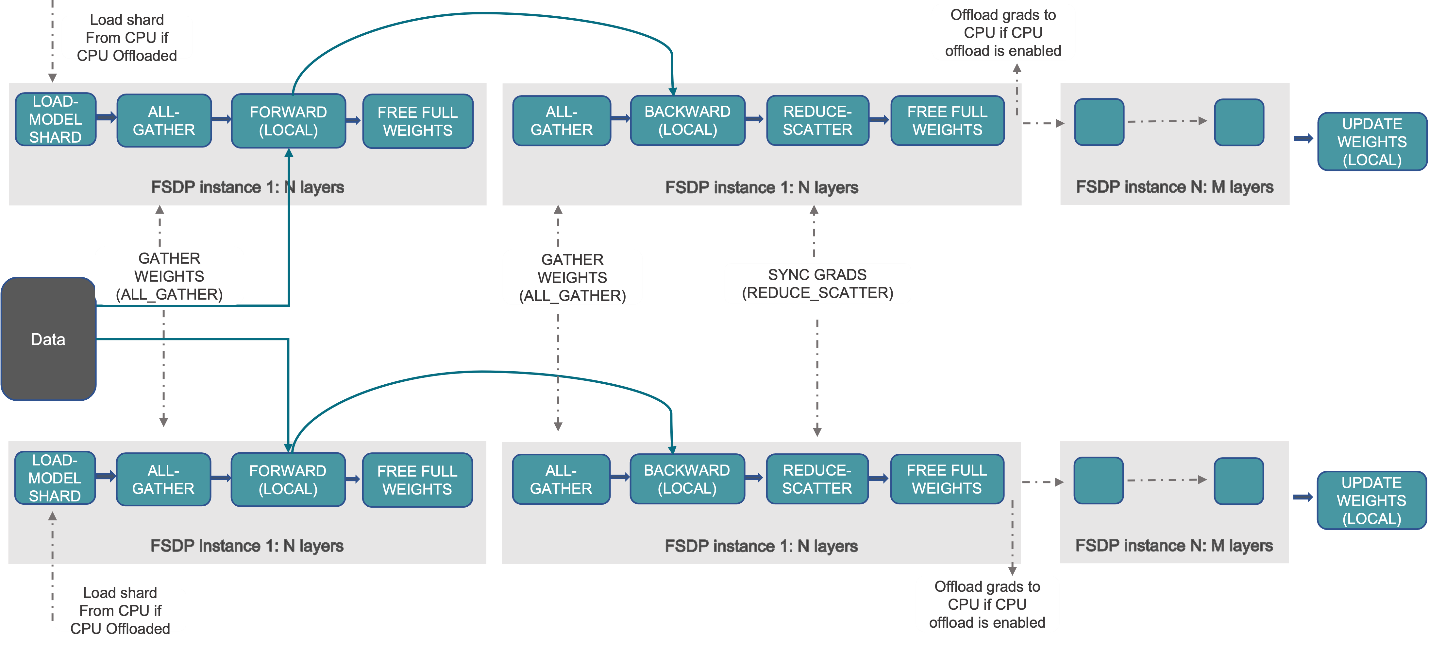
To make this happen, DDP registers an autograd hook for each parameter in the model. When the backward pass is run, this hook fires and triggers gradient synchronization across all processes. This ensures that each process has the same gradients, which are then used to update the model.

The recommended way to use DDP is to spawn one process for each model replica. The model replica can span multiple devices. DDP processes can be placed on the same machine or across machines. Note that GPU devices cannot be shared across DDP processes (i.e. one GPU for one DDP process).

## Fully Sharded Data Parallel (FSDP)

In [DistributedDataParallel](https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html), (DDP) training, each process/ worker owns a replica of the model and processes a batch of data, finally it uses all-reduce to sum up gradients over different workers. In DDP the model weights and optimizer states are replicated across all workers. FSDP is a type of data parallelism that shards model parameters, optimizer states and gradients across DDP ranks.

When training with FSDP, the GPU memory footprint is smaller than when training with DDP across all workers. This makes the training of some very large models feasible by allowing larger models or batch sizes to fit on device. This comes with the cost of increased communication volume. The communication overhead is reduced by internal optimizations like overlapping communication and computation.

[](https://pytorch.org/tutorials/_images/fsdp_workflow.png)

FSDP Workflow

At a high level FSDP works as follow:

*In constructor*

* Shard model parameters and each rank only keeps its own shard

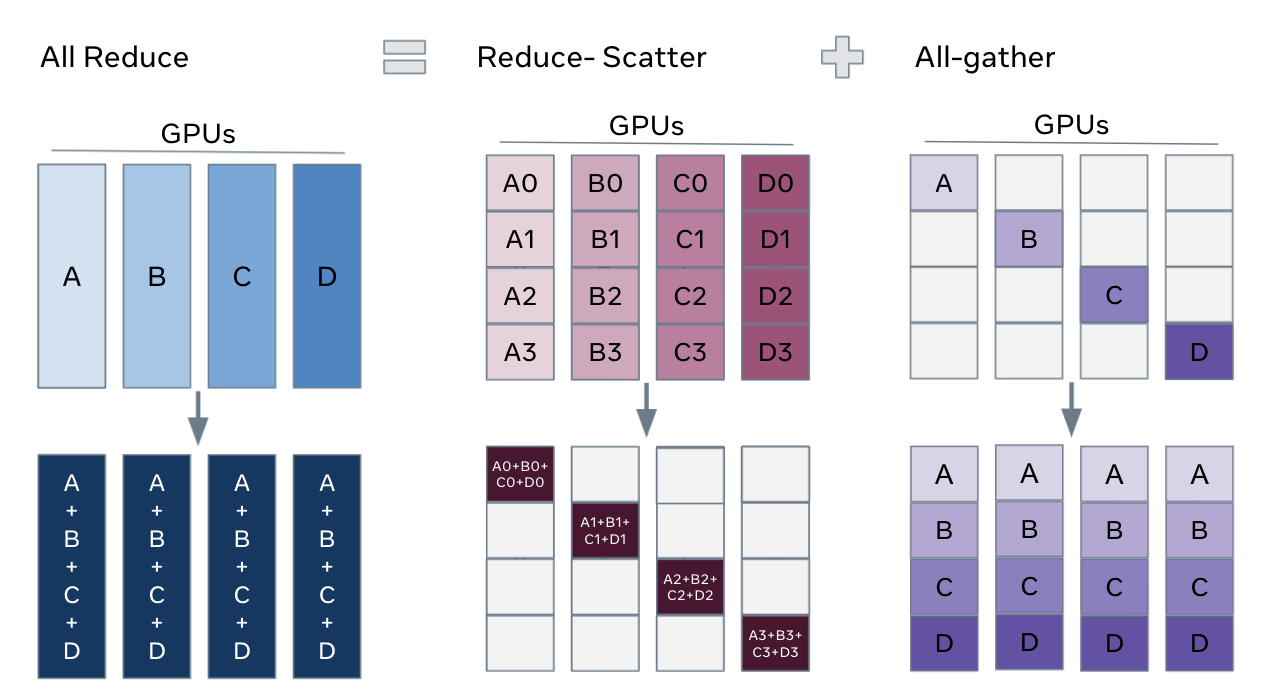
*In forward path*

* Run all\_gather to collect all shards from all ranks to recover the full parameter in this FSDP unit
* Run forward computation
* Discard parameter shards it has just collected

*In backward path*

* Run all\_gather to collect all shards from all ranks to recover the full parameter in this FSDP unit
* Run backward computation
* Run reduce\_scatter to sync gradients
* Discard parameters.

One way to view FSDP’s sharding is to decompose the DDP gradient all-reduce into reduce-scatter and all-gather. Specifically, during the backward pass, FSDP reduces and scatters gradients, ensuring that each rank possesses a shard of the gradients. Then it updates the corresponding shard of the parameters in the optimizer step. Finally, in the subsequent forward pass, it performs an all-gather operation to collect and combine the updated parameter shards.

[](https://pytorch.org/tutorials/_images/fsdp_sharding.png)