RankLLM: A Python Package for Reranking with LLMs

Sahel Sharifymoghaddam
sahel.sharifymoghaddam@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Ronak Pradeep
rpradeep@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Andre Slavescu
aslavesc@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Ryan Nguyen
ryan.nguyen@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Andrew Xu
a223xu@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Zijian Chen
s42chen@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Yilin Zhang
y2785zha@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Yidi Chen
e28chen@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Jasper Xian
jasper.xian@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

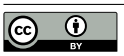
Jimmy Lin
jimmylin@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Abstract

The adoption of large language models (LLMs) as rerankers in multi-stage retrieval systems has gained significant traction in academia and industry. These models refine a candidate list of retrieved documents, often through carefully designed prompts, and are typically used in applications built on retrieval-augmented generation (RAG). This paper introduces RankLLM, an open-source Python package for reranking that is modular, highly configurable, and supports both proprietary and open-source LLMs in customized reranking workflows. To improve usability, RankLLM features optional integration with Pyserini for retrieval and provides integrated evaluation for multi-stage pipelines. Additionally, RankLLM includes a module for detailed analysis of input prompts and LLM responses, addressing reliability concerns with LLM APIs and non-deterministic behavior in Mixture-of-Experts (MoE) models. This paper presents the architecture of RankLLM, along with a detailed step-by-step guide and sample code. We reproduce results from RankGPT, LRL, RankVicuna, RankZephyr, and other recent models. RankLLM integrates with common inference frameworks and a wide range of LLMs. This compatibility allows for quick reproduction of reported results, helping to speed up both research and real-world applications. The complete repository is available at rankllm.ai, and the package can be installed via PyPI.

CCS Concepts

• **Information systems** → **Retrieval models and ranking**.



SIGIR '25, Padua, Italy

ACM ISBN 979-8-4007-1592-1/2025/07
https://doi.org/10.1145/3726302.3730331

Keywords

Information Retrieval; Reranking; Large Language Models; Python

ACM Reference Format:
Sahel Sharifymoghaddam, Ronak Pradeep, Andre Slavescu, Ryan Nguyen, Andrew Xu, Zijian Chen, Yilin Zhang, Yidi Chen, Jasper Xian, and Jimmy Lin. 2025. RankLLM: A Python Package for Reranking with LLMs. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '25), July 13–18, 2025, Padua, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3726302.3730331

1 Introduction

Pretrained transformer models have been applied to retrieval applications since the introduction of MonoBERT [33] in 2019. Over the past few years, the field has made significant progress, which can be grouped into two distinct families of techniques, known as cross-encoders and bi-encoders. Bi-encoders can be employed as first-stage retrievers, distinguishing themselves from cross-encoders, which are typically confined to reranking tasks. Together, they form complementary components in a multi-stage retrieval architecture [3, 7, 31, 36].

Recently, we have seen the introduction of a new class of reranking models, dubbed "prompt-decoders." Like cross-encoders, these approaches focus on reranking and are characterized by the use of modern large language models (LLMs), including proprietary models such as OpenAI's GPT_4 or open-source alternatives such as LLaMA, Vicuna, Zephyr, and Mistral. They rely heavily on prompt engineering and often leverage large context window sizes to implement listwise reranking. There has been an explosion of interest in this topic [29, 37–39, 42].

Although rerankers share many similarities, research in this space has largely depended on ad hoc and disparate implementations. This introduces complexity when comparing different approaches and impedes rapid exploration of the design space. The

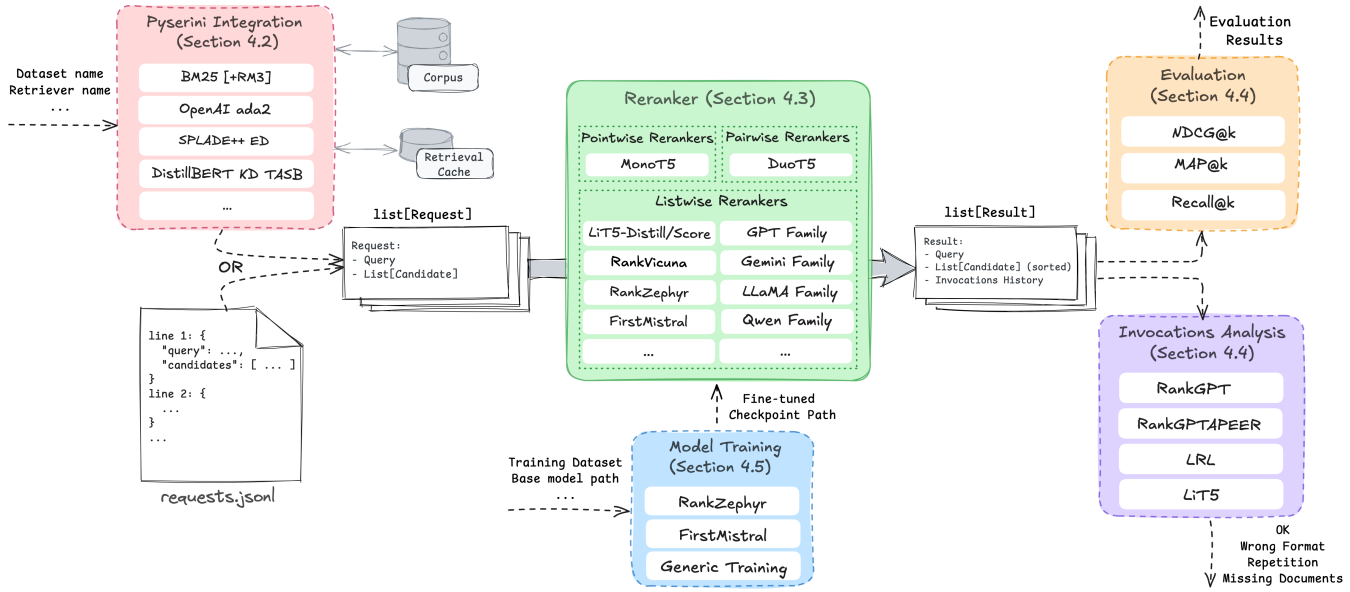


Figure 1: Overview of RankLLM with the Reranker component at the center. Other components facilitating optional flows include retrieval with Pyserini, evaluation, invocations analysis, and model training.

goal of this work is to alleviate these evident complexities. We present RankLLM, a Python package that facilitates reproducible training and evaluation of diverse rerankers, supporting community research and development in this area. More concretely, RankLLM, with its modular architecture (see Figure 1), offers the following:

- **Diverse model support** enables seamless integration of proprietary and open-source LLMs (MonoT5 [34], DuoT5 [36], RankVicuna [37], RankZephyr [38], LiT5 [43], FIRST [40], LRL [29], RankGPT [42], Gemini [44], and more) for pointwise, pairwise, and listwise reranking tasks. We also support most abstractions of the form RankX with the use of base open-source models through vLLM [23], enabling efficient serving and utilization of open-source LLMs for ranking purposes.
- **Modularity and configurability** customize reranking processes by allowing the selection of ranking methods, LLMs, inference frameworks, and prompt templates to suit specific requirements.
- **Auxiliary components for end-to-end flow** offer optional retrieval with Pyserini, invocations analysis, and evaluation tools. A training component is also available to fine-tune models like RankZephyr and FirstMistral for custom reranking needs when existing models do not suffice.
- **Reproducibility features** include predefined configurations, comprehensive logging, demo snippets, detailed docstrings and README instructions, and two-click reproducibility (2CR) [24] for transparent, reproducible results.

With this package, we aim to provide a valuable resource for the research community, laying the foundation for future advancements in effective, efficient, and reliable reranking models. Our work addresses the growing demand for high-quality reranking systems in the era of retrieval-augmented LLMs. RankLLM's expanding community adoption, as measured by the number of GitHub stars and PyPI downloads, highlights its impact and utility in modern AI pipelines. This growth is further supported by its integration with Rerankers [10], LlamaIndex [27], and LangChain [8], established frameworks for information retrieval (IR) and retrieval-augmented generation (RAG).

2 Background and Related Work

The primary goal in document retrieval is to identify, from a large corpus $C = \{D_1, D_2, \ldots, D_n\}$, a ranked subset of k documents that are most relevant to a given query q, where $k \ll |C|$. In this context, relevance is typically quantified using metrics such as normalized discounted cumulative gain (nDCG) or mean average precision (mAP). Modern retrieval systems generally follow a two-step procedure. The first stage consists of a fast and scalable retriever, which may employ sparse methods, like BM25 [41], or dense representations, best exemplified by DPR [22], to propose a candidate set of documents. In the subsequent reranking stage, more computationally expensive models — particularly transformer-based architectures — are applied to reassess and reorder the candidate list for enhanced effectiveness. Early transformer-based rerankers, such as MonoBERT [33], highlighted the benefits of joint query–document encoding in a cross-encoder setup, significantly outperforming traditional reranking methods [26]. Later studies introduced encoder–decoder models (e.g., MonoT5 [34], RankT5 [52]) and even decoder-only formulations [28] to further improve the reranking quality.

As shown in Figure 2, models for reranking have been organized around three paradigms: pointwise, pairwise, and listwise. In the pointwise formulation, each candidate document is evaluated independently with respect to the query, producing a relevance score that is then used to order the list. While straightforward, this approach does not capture the nuances of document interplay. Pairwise rerankers overcome this limitation by directly comparing pairs

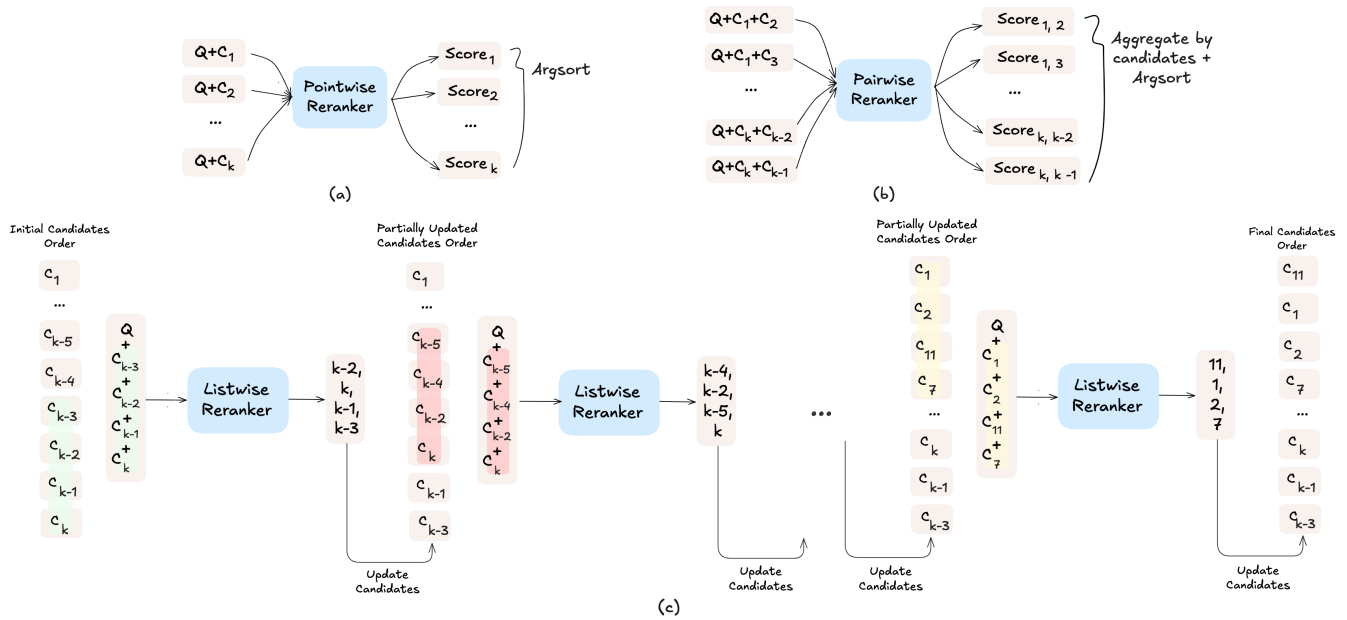


Figure 2: Three reranking methods applied to a Query Q and a list of k Candidates C_i: (a) pointwise reranking, (b) pairwise reranking, and (c) listwise reranking with a sliding window of size four and a stride of two.

of documents within the same context [33, 36], thus offering a relative judgment of relevance. More recently, listwise methods have emerged that directly process and reorder an entire candidate list in a single inference pass. These approaches, often termed "prompt-decoders" when implemented using LLMs, leverage the capabilities of instruction-tuned models to generate a complete permutation of the candidate documents based on relevance [29, 37, 38, 42, 49].

The listwise paradigm has gained momentum with the advent of powerful LLMs that can incorporate detailed prompt engineering to reason over multiple documents at once. For instance, RankGPT [42] demonstrated that using prompt-based listwise reranking can significantly enhance ranking effectiveness over traditional methods. Subsequent work has further refined this approach; models such as RankVicuna [37] distill the effectiveness of larger proprietary systems into smaller LLMs by leveraging instruction tuning, while RankZephyr [38] goes further, resulting in more effective and robust ranking than the teacher in several tasks. LiT5 [43] distills listwise behavior into smaller encoder–decoder models like T5, and results in a model capable of effectively ranking up to 100 segments at once. These models generate sequences denoting the reordering of the input documents. FirstMistral addresses generation inefficiencies by leveraging a learning-to-rank loss to train a model that can use the logits from the first token in the output sequence to determine the rank order of candidate documents, rather than generating a complete ranked sequence [9, 40]. One common challenge in listwise reranking is the limitation imposed by the LLM's input context window. Sliding window approaches process larger candidate lists in contiguous chunks, effectively balancing ranking quality with computational constraints [29, 38].

There is also a growing body of work that leverages LLMs for related data synthesis, used in some form by multi-stage retrieval pipelines. Examples include the synthesis of training data through query generation using techniques from InPars [5, 6], Promptagator [15], and PATH [46]. These data synthesis techniques, while primarily aimed at improving first-stage retrieval, highlight the broader trend of using LLMs to enhance various components of retrieval pipelines. Building on this momentum, recent work has increasingly explored the role of LLMs in more sophisticated reranking strategies.

In summary, while traditional reranking methods have relied on pointwise and pairwise scoring mechanisms, the emergence of LLM-based listwise rerankers represents a paradigm shift [19]. Our work builds on these recent advances by providing a modular and extensible Python package, RankLLM, that supports a variety of reranking models — from pointwise and pairwise to the more novel listwise approaches — while addressing practical challenges such as limited context sizes and non-deterministic model behavior. This integration of diverse reranking strategies within a reproducible, easy-to-use framework not only facilitates comprehensive experimentation in multi-stage retrieval systems but also underscores the evolving role of large language models in refining modern information retrieval pipelines.

3 Framework Overview

Figure 1 shows the high-level architecture of RankLLM with the Reranker as its main component. Each input `Request` to this component consists of a `Query` object with a "text" and a "qid", along with a list of relevant candidates (`list[Candidate]`). Each `Candidate` object in this list has a "docid", a "score" indicating its relevance to the query, and a "doc" dictionary with fields like "content", "title", etc. The output is a `Result` object that contains

```python
@dataclass
class Query:
    text: str
    qid: Union[str | int]

@dataclass
class Candidate:
    docid: Union[str | int]
    score: float
    doc: Dict[str, Any]

@dataclass
class Request:
    query: Query
    candidates: List[Candidate] = field(default_factory=list)

@dataclass
class InferenceInvocation:
    prompt: Any
    response: str
    input_token_count: int
    output_token_count: int

@dataclass
class Result:
    query: Query
    candidates: list[Candidate] = field(default_factory=list)
    invocations_history: list[InferenceInvocation] = ...
```

Figure 3: Data classes for reranking requests and results throughout the RankLLM pipeline. After reranking, the invocations history is also included in results.

```python
from dacite import from_dict

from rank_llm.data import Request, read_requests_from_file

# Create a request object inline
request_dict = {
    "query": {
        "text": "how long is life cycle of flea",
        "qid": "264014"
    },
    "candidates": [
        {
            "doc": {
                "segment": "The life cycle of a flea ..."
            },
            "docid": "4834547",
            "score": 14.97,
        },
        ...
    ],
}
request = from_dict(data_class=Request, data=request_dict)


# Load stored requests from a JSONL file
requests = read_requests_from_file(
    "retrieve_results/BM25/retrieve_results_dl23_top20.jsonl"
)
```

Figure 4: Sample requests creation either inline (lines 5–22) or via loading them from a JSONL file (lines 25–28).

the original query with a reordered list of candidates. Additionally, each `Result` object includes a list of `InferenceInvocation` objects, which store details of individual model inferences — such as the "prompt", "response", and "input/output_token_counts" — allowing for detailed analysis of the reranking process (Figure 3). Once the reranked results are available, users can use them directly for downstream tasks, store them in a file, or conduct their own evaluation. In addition to the main reranking component, RankLLM offers the following auxiliary components:

- **Retrieval** uses Pyserini [25] to retrieve relevant documents for a given query from a user-specified corpus using a user-specified retrieval method. It then stores them in the required `Request` format for the reranking step. This optional step provides an alternative to loading requests from a JSON(L) file or creating them inline.
- **Evaluation and analysis** can optionally evaluate the results using the TREC evaluation run format and analyze the reranking execution summary to identify malformed model responses for further investigation.
- **Training** supports distributed fine-tuning of LLMs through the Hugging Face Transformers library.[1] The framework supports multiple training objectives including the traditional language modeling objective and various learning-to-rank losses. For reproducibility, training recipes for models like RankZephyr and FirstMistral are also included.

[1]https://huggingface.co/docs/transformershuggingface.co/docs/transformers

4 RankLLM User Journey

This section covers installation, retrieval, various reranking methods, evaluation metrics, invocations analysis, and training, each explained in the following subsections.

4.1 Setup

Two options are available for obtaining and installing RankLLM:

PyPI Installation. RankLLM is available on the Python Package Index and it can be installed using the following pip command:

```
$ pip install rank-llm[all]
```

All code snippets in this paper use version 0.25.0. Please note that RankLLM is under active development, and the latest version at the time of reading may differ from the version referenced in this paper. However, we aim to ensure that all the snippets in the paper work as described.

Source Installation. RankLLM can be installed from the source repository, which is publicly available at rankllm.ai. This option is recommended for users who are interested in contributing to RankLLM. A `CONTRIBUTING.md` is included for onboarding volunteers.

```
$ git clone git@github.com:castorini/rank_llm.git
$ cd rank_llm
$ pip install -e .[all]
```

Both of these require a prior installation of CUDA-enabled PyTorch that works for the specific GPU configuration on hand. Once RankLLM is installed, users can follow along its retrieve and rerank pipeline as described in sections 4.2 to 4.5.

```python
from rank_llm.retrieve import RetrievalMethod, Retriever

# By default uses BM25 for retrieval
requests = Retriever.from_dataset_with_prebuilt_index("dl19")

# Users can specify other retrieval methods or customize the
# number of retrieved candidates per query.
requests = Retriever.from_dataset_with_prebuilt_index(
    dataset_name="dl20",
    retrieval_method=RetrievalMethod.SPLADE_P_P_ENSEMBLE_DISTIL,
    k=50,
)
```

Figure 5: Sample requests creation via Pyserini retrieval.

```python
from pathlib import Path

from rank_llm.data import DataWriter

# Write rerank results
writer = DataWriter(rerank_results)
Path("demo_outputs/").mkdir(parents=True, exist_ok=True)
writer.write_in_jsonl_format("demo_outputs/rerank_results.jsonl")
writer.write_in_trec_eval_format(
   "demo_outputs/rerank_results.txt"
)
writer.write_inference_invocations_history(
    "demo_outputs/inference_invocations_history.json"
)
```

Figure 6: Saving reranked results in JSON(L) and TREC run formats, and the execution summary in JSON format.

4.2 Retrieval and Caching

The `Reranker` accepts a single `Request` object or a list of them as input. As an alternative to creating them inline (lines 5–22 in Figure 4) or loading them from a JSON(L) file (lines 25–28 in Figure 4), users can optionally obtain them via the retriever component. The `Retriever` class in this component uses Pyserini with a user-specified dataset and retrieval method to find relevant candidates for each query, leveraging prebuilt indexes of the dataset corpus. It then converts the retrieved outputs into the required `Request` objects. BM25 [41] is the default retrieval method, retrieving the top 100 most relevant documents per query. However, users can specify other supported retrieval methods, including BM25+RM3 [2], OpenAI ada2 [32, 47], DistillBERT KD TASB [18], and SPLADE++ EnsembleDistil (ED) [17], or adjust the number of retrieved candidates per query (lines 6–12 in Figure 5).

Currently, the retriever supports the following datasets: (a) TREC 2019–2023 Deep Learning Tracks [11–14] (denoted DL19–DL23 in this paper); (b) all BEIR [45] datasets; (c) all languages from Mr.TyDi [50]. We plan to expand supported datasets as more prebuilt indexes become integrated into Pyserini. To reduce retrieval time, we cache the results of the most commonly asked queries. A complete list of cached results is available in the `repo_info` module under the `retrieve` sub-package.

4.3 Reranking

The `rerank` sub-package receives a list of `Request` objects and reorders the `Candidate` list for each object based on relevance to its `Query` using a RankLLM model coordinator. The output of this step is a list of `Result` objects where each result corresponds to a `Request` input object. To make result analysis simpler, the model coordinators also store all the created prompts and model responses for each request in its corresponding result. Once the rerank results are available, they can be either saved in specified files using the `DataWriter` helper class and/or analyzed using the next stage of the pipeline. As Figure 6 shows, the `DataWriter` produces the following auxiliary output files: (a) a JSON(L) file containing serialized rerank results; (b) a text file with each line in the standard TREC run format: `qid Q0 docno rank score tag`; (c) a JSON file containing serialized invocations history. Reranking as the core functionality of RankLLM supports the following model coordinator categories:

4.3.1 Pointwise. Pointwise coordinators individually score each candidate according to its relevance to the query. RankLLM integrates the family of MonoT5 [36] models for this purpose. Lines 20 and 21 in Figure 7 instantiate a MonoT5 model coordinator.

4.3.2 Pairwise. To take relative relevance into consideration, pairwise rerankers assign relevance scores to pairs of candidates. The final score per candidate is then calculated by aggregation. Similarly, RankLLM integrates the DuoT5 [36] model family as pairwise model coordinators (lines 23 and 24 in Figure 7).

4.3.3 Listwise. Listwise coordinators sort the list of candidates based on their relative relevance to the query by attending to them at once. RankLLM implements the following groups of listwise model coordinators:

- The LiT5 family [43] (lines 26–29 in Figure 7);
- Special-purpose listwise coordinators, such as `SafeOpenai` and `SafeGenai`, which use custom APIs for inference with specific proprietary models; in lines 31–35 from Figure 7, a `SafeOpenai` model coordinator is instantiated using the $\text{GPT}_{\text{4o-mini}}$ model and a context size of 4096—both required parameters. To use proprietary models, at least one valid API key is required.
- A generic `RankListwiseOSLLM` model coordinator that supports inference via vLLM, SGLang [51], or TensorRTLLM[2] APIs and is compatible with most popular open-source models on Hugging Face. For convenience, `VicunaReranker` and `ZephyrReranker` wrapper classes are provided to configure `RankListwiseOSLLM` model coordinators with default settings. Creating coordinators with these wrappers each requires only a single line of code (lines 51–53 in Figure 7).

Once a model coordinator is created, a `Reranker` object is simply initialized using it. This separation between the `Reranker` and its model coordinator allows the implementation of the rerank function to be delegated to the model coordinator while auxiliary methods, such as writing the reranking results, are maintained within the `Reranker` itself. For the first two coordinator categories, we refer to the original papers and repositories for implementation details. The rest of this section explains the implementation details of listwise model coordinators.

[2]https://github.com/NVIDIA/TensorRT-LLM

Sliding Window Algorithm. The number of retrieved candidates from the first stage is often too large to be included in a single prompt. To address this shortcoming, researchers have adopted a sliding window solution [28, 29, 37, 38, 42]. In this approach, the retrieved list is sent to the reranking model in chunks. With a window size of M, starting from the end of the candidate list, the last M documents are sent to the LLM and reordered. Then the window slides towards the beginning of the list by a stride of $N < M$ and the next M documents under the window are sent for reordering. This process continues until the window reaches the beginning of the list.

A single window pass from the end of a list of length k to its beginning brings the top $M - N$ candidates to the beginning of the list after $\lceil (k - M)/N \rceil$ calls to the model prediction. Since each pass of the sliding window from the end of the list to its front, partially orders the list, for a complete sorting of a list with length k, $k/(M - N)$ iterations of window passes are required.

A special case of the sliding window approach is bubble sort, in which k passes of a window of size two with a stride size of one are required to completely sort a given list. By default, RankLLM uses a single pass with a window size of 20 and a stride of 10 to rerank the top 100 candidates for each query. However, these parameters are all configurable as needed. During prompt creation, long passages are truncated if needed to fit within the specified context size.

Prompt Design. RankLLM uses an enum class to assign unique values to each prompt template. Each model coordinator can support a subset of these templates. During coordinator initialization, the user specifies the desired template. During reranking, the coordinator generates prompts according to the chosen template. This decoupled design simplifies the integration of new prompt templates in RankLLM. Currently, RankGPT [42], RankGPTAPEER [21], and LRL [29] prompt templates are implemented in the `SafeOpenai` model coordinator, with RankGPT as the default. `RankListwiseOSLLM` and `SafeGenai` also implement variations of RankGPT, while LiT5 uses its own prompt template.

By default, RankLLM prompts are zero-shot, meaning that no in-context examples are included in the prompt. However, users can specify the number of shots n and an input file for the example pool. The model coordinators will then randomly draw n examples from the pool and prepend them to the prompt. Each example in this case consists of a prompt and a response. All coordinators can include an optional system message at the beginning of the prompt.

Processing Invocation Responses. In listwise reranking, models return a reordered list of candidate ids after each invocation, and model coordinators update the candidates list accordingly. Since LLMs do not always respond in the expected format, coordinators gracefully process these responses to minimize error propagation. They first remove extra tokens, such as justifications or examples, then eliminate duplicate candidate ids, keeping only the first occurrence. Finally, any missing candidate ids are appended at the end, preserving their original order. This guarantees that the post-processed response is always a valid, sorted list of all candidate ids from the prompt.

```
1  from rank_llm.rerank import (
2      Reranker,
3      get_genai_api_key,
4      get_openai_api_key,
5  )
6  from rank_llm.rerank.listwise import (
7      RankListwiseOSLLM,
8      SafeGenai,
9      SafeOpenai,
10     VicunaReranker,
11     ZephyrReranker,
12 )
13 from rank_llm.rerank.pairwise.duot5 import DuoT5
14 from rank_llm.rerank.listwise.lit5_reranker import (
15     LiT5DistillReranker
16 )
17 from rank_llm.rerank.pointwise.monot5 import MonoT5
18
19
20 # Pointwise with MonoT5
21 reranker = Reranker(MonoT5("castorini/monot5-3b-msmarco-10k"))
22
23 # Pairwise with DuoT5
24 reranker = Reranker(DuoT5("castorini/duot5-3b-msmarco-10k"))
25
26 # Listwise with LiT5
27 reranker = Reranker(
28     LiT5DistillReranker("castorini/LiT5-Distill-large")
29 )
30
31 # Listwise with GPT models
32 model_coordinator = SafeOpenai(
33     "gpt-4o-mini", 4096, keys=get_openai_api_key(),
34 )
35 reranker = Reranker(model_coordinator)
36
37 # Listwise with Gemini models
38 model_coordinator = SafeGenai(
39     "gemini-2.0-flash-001", 4096, keys=get_genai_api_key()
40 )
41 reranker = Reranker(model_coordinator)
42
43 # Listwise with a generic model coordinator
44 model_coordinator = RankListwiseOSLLM(
45     model="castorini/first_mistral",
46     use_logits=True,
47     use_alpha=True,
48 )
49 reranker = Reranker(model_coordinator)
50
51 # Listwise with wrapper classes
52 reranker = VicunaReranker()
53 reranker = ZephyrReranker()
54
55 # Reranking
56 kwargs = {"populate_invocations_history": True}
57 rerank_results = reranker.rerank_batch(requests, **kwargs)
```

Figure 7: Reranking with different model coordinators: (a) pointwise (lines 20 and 21); (b) pairwise (lines 23 and 24); (c) listwise coordinators including LiT5 (lines 26–29), special-purpose coordinators for proprietary models (lines 31–41) and a generic coordinator (lines 43–49) for all open-source models. Wrapper classes simplify the creation of the generic coordinator for RankVicuna and RankZephyr (lines 51–53). Finally, the `reranker.rerank_batch()` is called to rerank.

```python
from rank_llm.analysis.response_analysis import ResponseAnalyzer
from rank_llm.evaluation.trec_eval import EvalFunction
from rank_llm.rerank import PromptMode
from rank_llm.retrieve.topics_dict import TOPICS


topics = TOPICS["dl19"]

# By default nDCG@10 is the eval metric.
ndcg10 = EvalFunction.from_results(rerank_results, topics)

# mAP@100
eval_args = ["-c", "-m", "map_cut.100", "-l2"]
map100 = EvalFunction.from_results(
  rerank_results, topics, eval_args
)

# recall@20
eval_args = ["-c", "-m", "recall.20"]
recall20 = EvalFunction.from_results(
  rerank_results, topics, eval_args
)

# Analyze response using RANK_GPT as the default template.
analyzer = ResponseAnalyzer.from_inline_results(rerank_results)
error_counts = analyzer.count_errors(verbose=True)

# Analyze response using the LRL template.
analyzer = ResponseAnalyzer.from_inline_results(
    rerank_results,
    prompt_mode=PromptMode.LRL,
)
error_counts = analyzer.count_errors(normalize=True)
```

Figure 8: Results evaluation and analysis

4.4 Evaluation and Analysis

As an optional step, the `Evaluation` sub-package quantifies the effectiveness of reranking by evaluating the `list[Result]` output. By default, it measures the nDCG@10 metric, but users can specify other metrics, such as mAP@100 or recall@20. As shown in Figure 8, query and relevance judgments (qrels) are required for evaluation. For convenience, the `retrieve` sub-package maintains a mapping from dataset names to their corresponding topics.

Since language models sometimes fail to follow prompt instructions and return malformed responses, users can optionally analyze LLM responses by using the `ResponseAnalyzer` class from the `analysis` sub-package. It counts the number of OK and malformed responses and returns a frequency dictionary. Following the common convention [37, 38, 42], malformed responses are categorized as: incorrect format, repetition or missing documents. This means the returned dictionary will have four keys: three for the error categories, and one for the OK responses.

4.5 Training

As an alternative to the existing reranking models, users have the option to train a custom reranker using the provided training module. This module supports fine-tuning of prompt-decoders through the Hugging Face Transformers library and leverages distributed training with Hugging Face Accelerate,[3] along with DeepSpeed Zero-3 for memory optimization.[4] To monitor the training process, we incorporate logging to Weights & Biases (W&B).[5] This module supports multiple training objectives, including traditional language modeling (LM) and various learning-to-rank losses, alongside regularization techniques such as NEFTune noisy embeddings [20]. For reproducibility, training recipes for models like RankZephyr and FirstMistral are also included.

```
accelerate launch train_rankllm.py \
    --model_name_or_path <path-to-model> \
    --train_dataset_path <path-to-train-dataset> \
    --num_train_epochs <num-epochs> \
    --seed <seed> \
    --per_device_train_batch_size <batch-size> \
    --num_warmup_steps <num-warmup-steps> \
    --gradient_checkpointing \
    --output_dir <output-directory> \
    --noisy_embedding_alpha <noisy-embedding-alpha> \
    --objective <objective>
```

Figure 9: Sample training run

Training Objectives. Traditional LM training optimizes for next-token prediction, which, as shown in FIRST [9, 40], is not inherently suited for reranking. Specifically, it penalizes misjudged documents at position 1 just as much as those at position 100, whereas more emphasis should be placed on top-ranked documents. To address this shortcoming, we support training with learning-to-rank losses, including RankNet, LambdaRank, and ListNet [40], which are better aligned with the reranking task. For instance, the RankNet loss is defined as:

$$\mathcal{L}_{\text{RankNet}} = \sum_{i=1}^{m} \sum_{j=1}^{m} \frac{\mathbf{1}_{r_i < r_j}}{i+j} \log(1 + \exp(s_i - s_j)) \tag{1}$$

where s_i and s_j are the model scores for documents i and j, and r_i and r_j are their relevance labels. This formulation places more penalty on errors at higher ranks (i.e., when $i + j$ is small). As demonstrated in [40], ranking effectiveness improves when using a combined objective:

$$\mathcal{L} = \mathcal{L}_{\text{LM}} + \lambda \mathcal{L}_{\text{Rank}} \tag{2}$$

where $\mathcal{L}_{\text{LM}}$ is the language modeling loss, $\mathcal{L}_{\text{Rank}}$ is one of the learning-to-rank losses, and λ is a hyperparameter controlling their relative weights.

Training Datasets. This component supports loading datasets from the Hugging Face Hub. By default, it uses datasets from RankZephyr, built from the MS MARCO V1 passage ranking training set with approximately 40K queries. For each query, the top 20 passages are retrieved using BM25 with Pyserini [25] and then reordered by a teacher model, GPT_4.

Training Recipes. Figure 9 illustrates a custom run of the training script in which `<path-to-model>` designates the base model to be fine-tuned, `<path-to-train-dataset>` identifies the training dataset, and `<objective>` defines the objective function. This function can be set to "generation" (using the standard LM objective), "ranking" (using a learning-to-rank approach), or "combined"

[3] https://huggingface.co/docs/acceleratehuggingface.co/docs/accelerate

[4] https://www.deepspeed.ai/tutorials/zero/deepspeed.ai/tutorials/zero

[5] https://wandb.ai/sitewandb.ai/site

	DL19	DL20	DL21	DL22	DL23
BM25 +	0.5058	0.4796	0.4458	0.2692	0.2624
(1) MonoT5	0.7174	0.6878	0.6678	0.4957	0.4505
(2) DuoT5	0.7302	0.6913	0.6931	0.5158	0.4600
(3a) LiT5Distill	0.7247	0.7049	0.6671	0.5102	0.4578
(3b) RankVicuna	0.6722	0.6551	0.6247	0.4352	0.4185
(3c) RankZephyr	0.7412	0.7106	0.7007	0.5111	0.4419
(3d) FirstMistral	0.7251	0.7009	0.6854	0.4889	0.4427
(4a) Qwen 2.5 7B Inst.	0.6784	0.6385	0.6465	0.4261	0.3889
(4b) LLaMA 3.1 8B Inst.	0.6688	0.6480	0.6573	0.4402	0.4018
(4c) Gemini Flash 2.0	0.7362	0.6930	0.6807	0.4805	0.4650
(4d) RankGPT *	0.7338	0.6792	0.6892	0.4884	0.4666
(4e) RankGPTAPEER *	0.7335	0.6925	0.6750	0.4786	0.4444
(4f) LRL *	0.7129	0.6564	0.6709	0.4729	0.4426

Table 1: nDCG@10 on DL19–DL23 datasets for reranking with (1) pointwise, (2) pairwise, (3*) specialized, and (4*) out-of-the-box listwise rerankers with BM25 used as the first-stage retrieval method. – * The $GPT_{4o\text{-}mini}$ model is used for all GPT runs with different prompt templates (rows 4d–4f).

(applying the hybrid objective described earlier). To support reproducibility, we include preset training recipes for both RankZephyr and FirstMistral. As an example, RankZephyr can be trained by executing the `scripts/train_rank_zephyr.sh` bash script from within the training directory.

5 Experimental Results

This section reproduces results from recent research on prompt-decoders [9, 21, 29, 37, 38, 42] using newer model versions where applicable, as well as rerankers recently added to RankLLM [16, 36, 43, 44]. We use DL19–DL23 datasets from the MS MARCO V1 and V2 [4] passage corpora as benchmarks and evaluate reranking effectiveness using nDCG@10. Table 1 shows nDCG@10 on DL19–DL23 datasets after reranking with (1) MonoT5; (2) DuoT5; (3*) specialized prompt-decoders; and (4*) out-of-the-box prompt-decoders. BM25 is used in the first stage to retrieve the top 100 candidates for each dataset. All experiments with open-source models are conducted using a single NVIDIA RTX A6000 GPU. As shown in RankVicuna [37], out-of-the-box prompt-decoders rank non-deterministically, meaning the same candidate list for a given query may be ranked slightly differently across runs. To save costs, we report single-run results, as comparing model effectiveness is not the main focus of this paper. For more accurate results, we recommend running each experiment multiple times.

All reported results are generated by running the end-to-end `demos/experimental_results.py` script in the repository. To further enhance reproducibility of RankLLM experiments, we provide 2CR reproduction pages.[6][7] In these pages the experimental conditions are aggregated into a reproduction matrix, allowing users to easily navigate through various settings and select those relevant to their interests or research needs. In these commands, `run_rank_llm.py` is a wrapper script that performs both the retrieval and reranking steps, encapsulating these operations within a simple command. By abstracting the underlying processes, the script enables users of all technical levels to reproduce our experimental setup and validate the findings, supporting our goals of transparency and open science. Table 2 shows the distribution of malformed responses from the same experiments. Although out-of-the-box prompt-decoders frequently generate malformed responses (e.g., between 28% to 75% of $GPT_{4o\text{-}mini}$ responses have missing candidate ids), graceful processing of these responses allows these models to still yield competitive results.

[6] https://castorini.github.io/rank_llm/src/rank_llm/2cr/msmarco-v1-passage.html
[7] https://castorini.github.io/rank_llm/src/rank_llm/2cr/msmarco-v2-passage.html

	OK(%)	Wrong Format(%)	Repetition(%)	Missing(%)
LiT5Distill	87.2	0.0	3.3	9.5
RankVicuna	100.0	0.0	0.0	0.0
RankZephyr	99.9	0.0	0.1	0.0
FirstMistral	99.9	0.0	0.0	0.1
Qwen 2.5 7B Inst.	71.2	2.4	26.4	0.0
LLaMA 3.1 8B Inst.	96.9	2.2	0.9	0.0
Gemini Flash 2.0	96.6	2.0	0.4	1.0
RankGPT *	63.4	8.3	0.1	28.2
RankGPTAPEER *	22.8	0.4	1.1	75.7
LRL *	59.0	12.1	0.1	28.8

Table 2: Distribution of malformed responses from prompt-decoders while reranking the top 100 candidates from DL19–DL23 datasets retrieved by BM25. – * The $GPT_{4o\text{-}mini}$ model is used for all GPT runs with different prompt templates.

6 Discussion

Several open-source tools focus on various stages of the retrieval, reranking, and RAG pipeline. Anserini [48] and Pyserini [25] specialize in retrieval, supporting both sparse and dense methods. PyTerrier [30] is designed for building IR pipelines with a focus on datasets and evaluation. PyGaggle [35] offers tools for neural reranking, but it primarily targets pointwise and pairwise reranking. Finally, a concurrent effort, Rankify [1] covers all three stages with extensions to question answering datasets and the Wikipedia corpus. While RankLLM focuses on reranking with prompt-decoders, it differentiates itself through lightweight integrations of complementary components, 2CR reproducibility, and support for training and invocation analysis. RankLLM intentionally stops at the reranking stage, leveraging its integration with established frameworks like LlamaIndex [27] and LangChain [8] for the final RAG stage.

RankLLM originated as a fork from the RankGPT repository [42], but we later decided to develop it as a standalone project, since implementing our desired functionalities required substantial code refactoring. Key new capabilities in RankLLM include: ranking with effective open-source prompt-decoders, pointwise and pairwise rerankers, support for various retrieval formats and caching, customizable prompt templates, integration with multiple inference frameworks, and 2CR reproducibility with invocation analysis.

RankLLM has evolved significantly with continuous feature enhancements. Recent updates include new datasets like DL23, integration with reranking models such as Gemini, LiT5, MonoT5, and the DuoT5 families, expanded support for prompt templates like RankGPTAPEER and FIRST, as well as training and batch inference support. By adding contribution guidelines and GitHub hooks for various sanity checks, we have accelerated feature development using community resources while maintaining a high-quality codebase. We believe RankLLM's modular design, thorough docstrings,

and automated test coverage make it easy to work with. Other design choices that contribute to RankLLM's extensibility and ensure highly reproducible results include:

Modular Structure. Splitting the RankLLM package into subpackages and modules enables the use of individual segments of the RankLLM pipeline. For example, users can use RankLLM solely for retrieval or provide their own retrieved results for reranking, skipping the retrieval stage entirely. Similarly, users can save reranked results to a file and skip the evaluation stage or bring their own rerank results for evaluation and invocations history analysis, bypassing the reranking stage.

Default Parametrization. Most parameters in RankLLM have default values, simplifying the main user experience while still allowing extensive customization.

Wrapper Classes. Whenever possible, wrapper classes abstract unnecessary details. For example, `VicunaReranker` and `ZephyrReranker` create `RankListwiseOSLLM` model coordinators using RankVicuna and RankZephyr models, respectively. However, users can create `RankListwiseOSLLM` model coordinators with any model compatible with vLLM, SGLang, or TensorRTLLM inference. Similar wrappers exist for creating `RankFiDDistill` and `RankFiDScore` model coordinators for the LiT5 family.

Extensible Reranking Framework. The `RankLLM` abstract class has three abstract sub-classes, each corresponding to pointwise, pairwise, and listwise reranking. Any new reranking model coordinator can inherit from one of these classes and implement its public abstract methods, simplifying model integration.

Efficient Retrieval Caching and Storage. The retriever stores retrieved results in files named based on retrieval parameters. This allows RankLLM to reload results from stored files when the same retrieval parameters are used. Additionally, structured naming enables caching for common retrieval queries systematically.

Systematic Output Organization. The `Rerank` class has a helper function that saves results using a structured and informative naming convention. Each output filename encodes key reranking parameters, such as retrieval method, reranking model, context length, number of candidates, dataset, and a timestamp indicating when the run was executed. This systematic approach to naming simplifies both result analysis and file management, allowing users to easily trace outputs back to their configurations.

7 Conclusion

We introduced RankLLM, an open-source Python package which reranks candidate lists based on their relevance to a given query. RankLLM supports listwise reranking with GPT family models from OpenAI, Gemini family models from Google, LiT5 family models, and any LLM compatible with vLLM, SGLang, or TensorRT-LLM inference frameworks. For completeness, MonoT5 and DuoT5 models are also integrated for pointwise and pairwise reranking, respectively. RankLLM relies on default parameters to simplify common reranking tasks while still providing configurable options for custom use cases. It is designed to be modular, configurable, and user-friendly, enabling researchers and practitioners to experiment with various reranking strategies in multi-stage retrieval systems. To support an easy end-to-end workflow, RankLLM also includes complementary components for training, retrieval, evaluation, and invocation analysis.

With comprehensive documentation, demo scripts, and support for a wide range of language models and prompt templates, RankLLM facilitates rapid prototyping and experimentation. Its emphasis on reproducibility—achieved through detailed logging and two-click reproduction pages—ensures that experiments can be easily shared, validated, and extended by the broader research community. By lowering the barrier to experimenting with reranking methods, RankLLM aims to accelerate innovation and foster collaboration in the growing fields of information retrieval and large language models. Integration of RankLLM into well-known ranking and RAG frameworks like Rerankers, LangChain, and LlamaIndex, along with its PyPI downloads and repository stars, highlight its widespread adoption and community impact.

Acknowledgments

This research was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada. Additional funding was provided by Microsoft via the Accelerating Foundation Models Research program and an Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean Government (MSIT) (No. RS-2024-00457882, National AI Research Lab Project). We would like to thank Lily Ge and Daniel Guo for their thoughtful comments. We also thank Akintunde Oladipo, Patrick Yi, Nathan Gabriel Kuissi, Charlie Liu, Brayden Zhong and all other community contributors for their valuable work on the open-source repository of RankLLM.

References

[1] Abdelrahman Abdallah, Jamshid Mozafari, Bhawna Piryani, Mohammed Ali, and Adam Jatowt. 2025. Rankify: A Comprehensive Python Toolkit for Retrieval, Re-Ranking, and Retrieval-Augmented Generation. *arXiv:2502.02464* (2025).

[2] Nasreen Abdul-Jaleel, James Allan, W. Bruce Croft, Fernando Diaz, Leah Larkey, Xiaoyan Li, Donald Metzler, Mark D. Smucker, Trevor Strohman, Howard Turtle, and Courtney Wade. 2004. UMass at TREC 2004: Novelty and HARD. In *Proceedings of the Thirteenth Text REtrieval Conference (TREC 2004)*. Gaithersburg, Maryland.

[3] Nima Asadi and Jimmy Lin. 2013. Effectiveness/Efficiency Tradeoffs for Candidate Generation in Multi-Stage Retrieval Architectures. In *Proceedings of the 36th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2013)*. Dublin, Ireland, 997–1000.

[4] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. 2016. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. *arXiv:1611.09268v3* (2016).

[5] Luiz Bonifacio, Hugo Abonizio, Marzieh Fadaee, and Rodrigo Nogueira. 2022. InPars: Unsupervised Dataset Generation for Information Retrieval. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2022)*. Madrid, Spain, 2387–2392.

[6] Leonid Boytsov, Preksha Patel, Vivek Sourabh, Riddhi Nisar, Sayani Kundu, Ramya Ramanathan, and Eric Nyberg. 2023. InPars-Light: Cost-Effective Unsupervised Training of Efficient Rankers. *arXiv:2301.02998* (2023).

[7] B. Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. 2010. Early Exit Optimizations for Additive Machine Learned Ranking Systems. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining (WSDM 2010)*. New York, New York, 411–420.

[8] Harrison Chase. 2022. *LangChain*.

[9] Zijian Chen, Ronak Pradeep, and Jimmy Lin. 2024. An Early FIRST Reproduction and Improvements to Single-Token Decoding for Fast Listwise Reranking. *arXiv:2411.05508* (2024).

[10] Benjamin Clavié. 2024. Rerankers: A Lightweight Python Library to Unify Ranking Methods. *arXiv:2408.17344* (2024).

[11] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, and Daniel Campos. 2021. Overview of the TREC 2020 Deep Learning Track. *arXiv:2102.07662* (2021).

[12] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, Daniel Campos, and Jimmy Lin. 2022. Overview of the TREC 2021 Deep Learning Track. In *Proceedings of the Thirtieth Text REtrieval Conference (TREC 2021)*.

[13] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, Daniel Campos, Jimmy Lin, Ellen M. Voorhees, and Ian Soboroff. 2023. Overview of the TREC 2022 Deep Learning Track. In *Proceedings of the Thirty-First Text REtrieval Conference (TREC 2022)*.

[14] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, Daniel Campos, and Ellen M. Voorhees. 2020. Overview of the TREC 2019 Deep Learning Track. *arXiv:2003.07820* (2020).

[15] Zhuyun Dai, Vincent Zhao, Ji Ma, Yi Luan, Jianmo Ni, Jing Lu, Anton Bakalov, Kelvin Guu, Keith B. Hall, and Ming-Wei Chang. 2022. Promptagator: Few-shot Dense Retrieval From 8 Examples. *arXiv:2209.11755* (2022).

[16] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The LLaMA 3 Herd of Models. *arXiv:2407.21783* (2024).

[17] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2022. From Distillation to Hard Negative Sampling: Making Sparse Neural IR Models More Effective. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2022)*. Madrid, Spain, 2353–2359.

[18] Sebastian Hofstätter, Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and Allan Hanbury. 2021. Efficiently Teaching an Effective Dense Retriever with Balanced Topic Aware Sampling. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*. 113–122.

[19] Mathew Jacob, Erik Lindgren, Matei Zaharia, Michael Carbin, Omar Khattab, and Andrew Drozdov. 2024. Drowning in Documents: Consequences of Scaling Reranker Inference. *arXiv:2411.11767* (2024).

[20] Neel Jain, Ping-yeh Chiang, Yuxin Wen, John Kirchenbauer, Hong-Min Chu, Gowthami Somepalli, Brian R. Bartoldson, Bhavya Kailkhura, Avi Schwarzschild, Aniruddha Saha, et al. 2023. NEFTune: Noisy Embeddings Improve Instruction Finetuning. *arXiv:2310.05914* (2023).

[21] Can Jin, Hongwu Peng, Shiyu Zhao, Zhenting Wang, Wujiang Xu, Ligong Han, Jiahui Zhao, Kai Zhong, Sanguthevar Rajasekaran, and Dimitris N. Metaxas. 2024. APEER: Automatic Prompt Engineering Enhances Large Language Model Reranking. *arXiv:2406.14449* (2024).

[22] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6769–6781.

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with Paged-Attention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[24] Jimmy Lin. 2022. Building a Culture of Reproducibility in Academic Research. *arXiv:2212.13534* (2022).

[25] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A Python Toolkit for Reproducible Information Retrieval Research with Sparse and Dense Representations. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*. 2356–2362.

[26] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. 2021. *Pretrained Transformers for Text Ranking: BERT and Beyond*. Morgan & Claypool Publishers.

[27] Jerry Liu. 2022. *LlamaIndex*.

[28] Xueguang Ma, Liang Wang, Nan Yang, Furu Wei, and Jimmy Lin. 2024. Fine-tuning LLaMA for Multi-Stage Text Retrieval. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2024)*. 2421–2425.

[29] Xueguang Ma, Xinyu Zhang, Ronak Pradeep, and Jimmy Lin. 2023. Zero-Shot Listwise Document Reranking with a Large Language Model. *arXiv:2305.02156* (2023).

[30] Craig Macdonald and Nicola Tonellotto. 2020. Declarative Experimentation inInformation Retrieval using PyTerrier. In *Proceedings of the 2020 ACM SIGIR on International Conference on Theory of Information Retrieval (ICTIR 2020)*.

[31] Irina Matveeva, Chris Burges, Timo Burkard, Andy Laucius, and Leon Wong. 2006. High Accuracy Retrieval with Multiple Nested Ranker. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006)*. Seattle, Washington, 437–444.

[32] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, et al. 2022. Text and Code Embeddings by Contrastive Pre-Training. *arXiv:2201.10005* (2022).

[33] Rodrigo Nogueira and Kyunghyun Cho. 2019. Passage Re-ranking with BERT. *arXiv:1901.04085* (2019).

[34] Rodrigo Nogueira, Zhiying Jiang, Ronak Pradeep, and Jimmy Lin. 2020. Document Ranking with a Pretrained Sequence-to-Sequence Model. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 708–718.

[35] Ronak Pradeep, Haonan Chen, Lingwei Gu, Manveer Singh Tamber, and Jimmy Lin. 2023. PyGaggle: A Gaggle of Resources for Open-Domain Question Answering. In *Advances in Information Retrieval: 45th European Conference on Information Retrieval (ECIR 2023), Part III*. Dublin, Ireland, 148–162.

[36] Ronak Pradeep, Rodrigo Nogueira, and Jimmy Lin. 2021. The Expando-Mono-Duo Design Pattern for Text Ranking with Pretrained Sequence-to-Sequence Models. *arXiv:2101.05667* (2021).

[37] Ronak Pradeep, Sahel Sharifymoghaddam, and Jimmy Lin. 2023. RankVicuna: Zero-shot Listwise Document Reranking with Open-Source Large Language Models. *arXiv:2309.15088* (2023).

[38] Ronak Pradeep, Sahel Sharifymoghaddam, and Jimmy Lin. 2023. RankZephyr: Effective and Robust Zero-Shot Listwise Reranking is a Breeze! *arXiv:2312.02724* (2023).

[39] Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Le Yan, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, Xuanhui Wang, and Michael Bendersky. 2024. Large Language Models are Effective Text Rankers with Pairwise Ranking Prompting. In *Findings of the Association for Computational Linguistics: NAACL 2024*. Mexico City, Mexico, 1504–1518.

[40] Revanth Gangi Reddy, JaeHyeok Doo, Yifei Xu, Md Arafat Sultan, Deevya Swain, Avirup Sil, and Heng Ji. 2024. FIRST: Faster Improved Listwise Reranking with Single Token Decoding. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Miami, Florida, USA, 8642–8652.

[41] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval* 3, 4 (2009), 333–389.

[42] Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. 2023. Is ChatGPT Good at Search? Investigating Large Language Models as Re-Ranking Agents. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 14918–14937.

[43] Manveer Singh Tamber, Ronak Pradeep, and Jimmy Lin. 2023. Scaling Down, LiTting Up: Efficient Zero-Shot Listwise Reranking with Seq2seq Encoder-Decoder Models. *arXiv:2312.16098* (2023).

[44] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking Multimodal Understanding Across Millions of Tokens of Context. *arXiv:2403.05530* (2024).

[45] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Vol. 1.

[46] Jasper Xian, Saron Samuel, Faraz Khoubsirat, Ronak Pradeep, Md Arafat Sultan, Radu Florian, Salim Roukos, Avirup Sil, Christopher Potts, and Omar Khattab. 2024. Prompts as Auto-Optimized Training Hyperparameters: Training Best-in-Class IR Models from Scratch with 10 Gold Labels. *arXiv:2406.11706* (2024).

[47] Jasper Xian, Tommaso Teofili, Ronak Pradeep, and Jimmy Lin. 2024. Vector Search with OpenAI Embeddings: Lucene is All You Need. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*. New York, NY, USA, 1090–1093.

[48] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *Proceedings of the 40th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2017)*. Tokyo, Japan, 1253–1256.

[49] Crystina Zhang, Sebastian Hofstätter, Patrick Lewis, Raphael Tang, and Jimmy Lin. 2025. Rank-Without-GPT: Building GPT-Independent Listwise Rerankers on Open-Source Large Language Models. In *Advances in Information Retrieval*. Cham, 233–247.

[50] Xinyu Zhang, Xueguang Ma, Peng Shi, and Jimmy Lin. 2021. Mr. TyDi: A Multilingual Benchmark for Dense Retrieval. In *Proceedings of the 1st Workshop on Multilingual Representation Learning*. Punta Cana, Dominican Republic, 127–137.

[51] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems*, Vol. 37. 62557–62583.

[52] Honglei Zhuang, Zhen Qin, Rolf Jagerman, Kai Hui, Ji Ma, Jing Lu, Jianmo Ni, Xuanhui Wang, and Michael Bendersky. 2023. RankT5: Fine-tuning T5 for Text Ranking with Ranking Losses. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2023)*. 2308–2313.