LLM Ensembling: Haystack Pipelines with LLM-Blender

Ashwin Mathur* Varun Mathur*
{ashwinxmathur, varunm500}@gmail.com

July 22, 2024

Abstract

LLM-Blender is an ensembling framework designed to achieve consistently superior performance by combining the outputs of multiple language models (LLMs). This work focuses on integrating LLM-Blender with Retrieval-Augmented Generation (RAG) pipelines to significantly improve the quality of generated text. A custom Haystack component, *LLMBlenderRanker*, has been implemented to integrate LLM-Blender with Haystack pipelines. The component utilizes the *PairRanker* module from the LLM-Blender framework, which compares each candidate with the input in a pairwise manner. Different LLMs can generate subtly different texts, since they are trained on different datasets and tasks. By comparing each text in a pairwise manner, the component ranks and ensembles the text so it is robust to these subtle differences. Ranking techniques like MLM-scoring, SimCLS, and SummaReranker focus on individually scoring each candidate based on the input text, but do not compare candidates in a pairwise manner, which can lead to missing subtle differences between LLM outputs. Pipelines ensembling various LLMs, such as Mistral-7B, Llama-3-8B and Phi-3-mini, using the LLM-Blender were evaluated. The MixInstruct benchmark dataset was curated by the LLM-Blender authors to benchmark ensemble models for LLMs on instruction-following tasks. The pipelines were evaluated using the BERTScore, BARTScore, and BLEURT metrics on the MixInstruct and BillSum Datasets. The LLM-Blender framework, also introduced a *GenFuser* module, which fuses multiple LLM outputs to give a single condensed result. We found that the usage of the PairRanker alone gives better performance in RAG pipelines, as compared to using the PairRanker and GenFuser together. We obtained better results on the MixInstruct and BillSum datasets than the results presented in the LLM Blender paper using more recent LLMs such as Mistral-7B, Llama-3-8B and Phi-3-mini.

1 Introduction

There are many open-source Large Language Models (LLMs), each having it's strengths and weaknesses. Each LLM is trained using different data, hyperparameters, and architectures. The performance of these LLM's also differs, there does not exist a single open-source LLM which dominates. This leads to the idea of ensembling different open-source LLM's, in order to benefit from the strengths of each LLM.

LLM-Blender (D. Jiang, Ren, and B. Y. Lin 2023) lets you ensemble the outputs from multiple LLMs to get the best of each LLM. LLM-Blender is able to achieve superior performance by mixing the outputs of multiple LLMs. It does this by ranking and fusing the outputs from multiple LLMs. We set out to investigate the performance of Haystack RAG pipelines using LLM-Blender, and our results are very encouraging.

The rest of this paper is organized as follows: Section 2 of this paper explains the two components of the LLM-Blender, the PairRanker and the GenFuser. Section 3 describes the datasets used in the experiments with the LLM-Blender. Section 4 describes the LLMs which are being used in the pipelines. Section 5 describes the metrics used to evaluate the pipelines. Section 6 describes the integration of LLM-Blender with Haystack by creating a custom LLMBlenderRanker component. Section 7 explains how Haystack Pipelines are used with LLM-Blender, and includes the experiments done on the BillSum and MixInstruct datasets using the LLMBlenderRanker component. Section 8 contains a detailed analysis of results obtained in the experiments.

*Equal Contribution

2 LLM-Blender

LLM-Blender is an ensembling framework designed to achieve consistently superior performance by combining the outputs of multiple LLM's. The LLM-Blender framework can be used to enhance the outputs generated by multiple Retrieval Augmented Generation Pipelines (RAG) (P. Lewis et al. 2020), which leverage diverse open-source Large Language Models (LLMs).

LLM-Blender is a two-stage ensemble learning framework. In the first stage (ranking), pairwise comparison of candidates is performed, and they are then ranked. In the second stage (fusing), the top K candidates are merged to render the final output.

The LLM-Blender comprises of two modules: the PairRanker and the GenFuser. The PairRanker module compares the outputs from multiple LLMs to provide the top-ranked outputs. It compares each candidate with the input in a pairwise manner, making it robust to subtle differences in the generated text. The GenFuser module uses the top-ranked outputs from the PairRanker module to generate an improved output. The module fuses the top K of the N-ranked candidates from the PairRanker, conditioned on the input instruction, to generate an enhanced output.

2.1 PairRanker

The PairRanker module is responsible for comparing and ranking the outputs from LLM's. During the ranking stage, a specific input prompt (x) is passed to N different LLMs, and their outputs are compiled as candidates ($y_1, \ldots, y_N$).

The PairRanker then analyzes and ranks these candidates. For each input x, the candidates are obtained from N different LLMs. This input sequence, along with the candidates, is then subjected to a cross-attention text encoder, such as RoBERTa. The text encoder is tasked with learning and determining the superior candidate for the given input x. All the candidates are paired (y_i and y_j), producing a matrix of pairwise comparison results. These pairs are evaluated based on the condition: given the input prompt, which candidate's output is better? By aggregating the results in the matrix, the PairRanker can rank all candidates and take the top K of them for generative fusion.

The effectiveness of the PairRanker is constrained by the quality of selections from the candidate pool. Alternatively, ranking techniques like MLM-scoring (Salazar et al. 2019), SimCLS (Y. Liu and P. Liu 2021) and SummaReranker (Ravaut, Joty, and Chen 2022) give ranked outputs with an absolute score. In such cases, the top-k ranked outputs from these techniques are used for analysis.

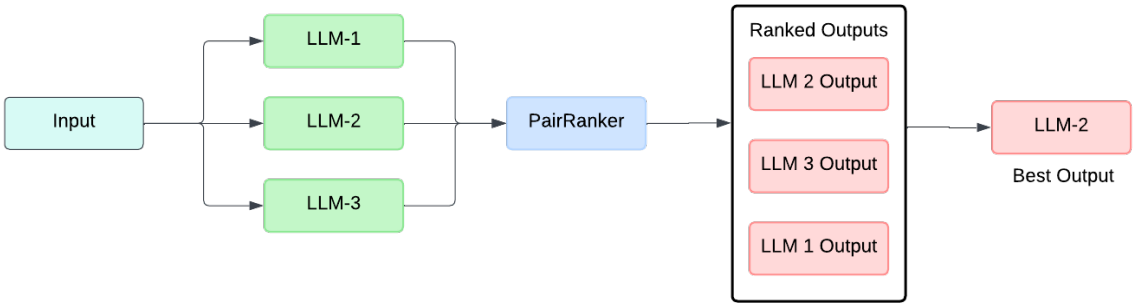

Figure 1: Pipeline with a PairRanker

2.2 GenFuser

The primary goal of the GenFuser module is to capitalize on the strengths of the top K selected candidates from the PairRanker's ranking.

After the PairRanker module ranks the candidates, the GenFuser module is employed to fuse the top K out of the N ranked candidates and generate an improved final output. It takes a seq2seq approach, fusing the set of top candidates while conditioning on the input prompt, to generate an improved and enhanced output.

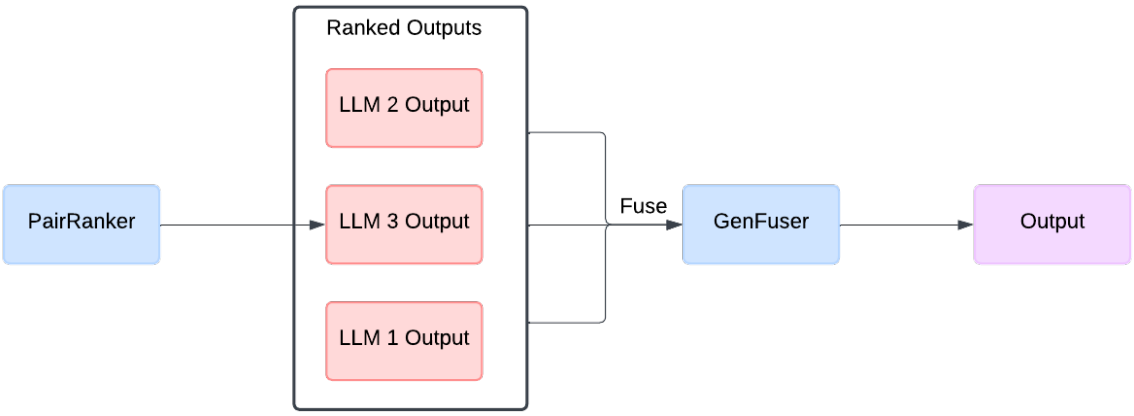

Figure 2: Pipeline with a PairRanker

3 Mixture-of-Agents (MoA) to enhance LLM Capabilities

- LLMs are pretrained on vast amounts of data and subsequently aligned with human preferences to generate helpful and coherent outputs. However, despite the plethora of LLMs and their impressive achievements, they still face inherent constraints on model size and training data.
- Further scaling up these models is exceptionally costly, often requiring extensive retraining on several trillion tokens.
- At the same time, different LLMs possess unique strengths and specialize in various tasks aspects. For instance, some models excel at complex instruction following while others may be better suited for code generation.
- The Mixture-of-Agents (MoA) methodology (J. Wang et al. 2024) leverages multiple LLMs to iteratively enhance the generation quality. This is done by utilizing multiple LLMs (agents) to independently generate responses via successive stages for iterative collaboration.

Selection of Models for MoA layers:

To ensure effective collaboration among models and improve overall response quality, careful selection of LLMs for each MoA layer is crucial. This selection process is guided by two primary criteria:

- Performance Metrics: The average win rate of models in layer i plays a significant role in determining their suitability for inclusion in layer i + 1. Therefore, selecting models based on their demonstrated performance metrics ensures higher-quality outputs.
- Diversity Considerations: The diversity of model outputs is also crucial. Responses generated by heterogeneous models contribute significantly more than those produced by the same model.

Structure of MoA:

The Mixture-of-Agents (MoA) methodology relies on multiple layers of LLMs for iterative refinement of the responses.

- Initially, LLMs in the first layer, denoted as agents A1,1, ...A1,n independently generate responses to a given prompt.
- These responses are then presented to agents in the next layer A2,1, ...A2,n (which may reuse a model from the first layer) for further refinement.
- This iterative refinement process continues for several cycles until obtaining a more robust and comprehensive response.

- An important pathway to extract maximum benefits from collaboration of multiple LLMs is to characterize how different models are good at in various aspects of collaboration. During the collaboration process, we can categorize LLMs into two distinct roles: Proposers and Aggregators.
- Proposers excel at generating useful reference responses for use by other models. While a good proposer may not necessarily produce responses with high scores by itself, it should offer more context and diverse perspectives, ultimately contributing to better final responses when used by an aggregator.
- Aggregators are models proficient in synthesizing responses from other models into a single, high-quality output. An effective aggregator should maintain or enhance output quality even when integrating inputs that are of lesser quality than its own.
- Many LLMs possess capabilities both as aggregators and proposers, while certain models displayed specialized proficiencies in distinct roles. GPT-4o, Qwen1.5, LLaMA-3 emerged as a versatile model effective in both assisting and aggregating tasks. In contrast, WizardLM demonstrated excellent performance as an proposer model but struggled to maintain its effectiveness in aggregating responses from other models.
- This is replicated with multiple aggregators, initially using several to aggregate better answers and then re-aggregating these aggregated answers. By incorporating more aggregators into the process, we can iteratively synthesize and refine the responses, leveraging the strengths of multiple models

4 DSPy

- LLMs are known to be sensitive to how they are prompted for each task, and this is exacerbated in pipelines where multiple LM calls have to interact effectively.
- As a result, the LLM calls in existing LLM pipelines and in popular developer frameworks are generally implemented using hard-coded 'prompt templates', that is, long strings of instructions and demonstrations that are hand crafted through manual trial and error.
- This approach can be brittle and unscalable—conceptually akin to hand-tuning the weights for a classifier. A given string prompt might not generalize to different pipelines or across different LLMs, data domains, or even inputs.
- DSPy pushes building LLM pipelines away from manipulating free-form strings and closer to programming (composing modular operators to build text transformation graphs) where a compiler automatically generates optimized LLM invocation strategies and prompts from a program.
- The DSPy framework consists of two components: **DSPy Programming Model** and **DSPy Compiler**.
- The DSPy programming model translates string-based prompting techniques, including complex and task-dependent ones like Chain of Thought and ReAct, into declarative modules that carry natural-language typed signatures.
- The modules are task-adaptive components—akin to neural network layers—that abstract any particular text transformation, like answering a question or summarizing a paper.
- Each DSPy module is parameterized so that it can learn its desired behavior by iteratively bootstrapping useful demonstrations within the pipeline. Inspired directly by PyTorch abstractions, DSPy modules are used via expressive define-by-run computational graphs.
- The pipelines are expressed by (1) declaring the modules needed and (2) using these modules in any logical control flow (e.g., if statements, for loops, exceptions, etc.) to logically connect the modules.
- The DSPy compiler optimizes any DSPy program to improve quality or cost. The compiler inputs are the program, a few training inputs with optional labels, and a validation metric.
- The compiler simulates versions of the program on the inputs and bootstraps example traces of each module for self-improvement, using them to construct effective few-shot prompts or finetuning small LLMs for steps of the pipeline.

- Optimization in DSPy is highly modular: it is conducted by *teleprompters*, which are general-purpose optimization strategies that determine how the modules should learn from data.
- In this way, the compiler automatically maps the declarative modules to high-quality compositions of prompting, finetuning, reasoning, and augmentation.

Ensembling and Self-Improving Pipelines using DSPy:

- DSPy programs are expressed in Python: each program takes the task input (e.g., a question to answer or a paper to summarize) and returns the output (e.g., an answer or a summary) after a series of steps.
- DSPy uses three abstractions toward automatic optimization: *signatures*, *modules*, and *teleprompters*.
- Signatures abstract the input/output behavior of a module; modules replace existing hand-prompting techniques and can be composed in arbitrary pipelines; and teleprompters optimize all modules in the pipeline to maximize a metric.
- Solving complex tasks with LLMs requires applying sophisticated prompting techniques and chaining them together into multi-stage pipelines.
- However, LLM programs today are commonly designed via "prompt engineering": crafting lengthy prompts via manual trial and error to coerce a specific LLM to operate each step in a specific pipeline.
- Multi-Stage pipelines are improved using a 3 step process: (i) construct instructive example traces (i.e. inputs and outputs for each module) that demonstrate how the LLMs should conduct the task, (ii) collect and summarize important factors that could shape the construction of high-quality instructions, and/or (iii) meta-optimize how LLMs are used to propose high-performing instructions.
- Optimizing bootstrapped demonstrations as few-shot examples is key to achieving the best performance.
- Instruction optimization is most important for tasks with conditional rules that are (i) not immediately obvious to the LM and (ii) not expressible via a limited number of few-shot examples.
- Grounding is helpful for instruction proposal, but the best proposal strategy varies by task. Across tasks, the highest importance scores go to the choice of bootstrapped demonstrations in the meta-prompt and the tip. The importance of many parameters varies between tasks.

5 Datasets used in RAG Pipelines

For evaluating the performance of RAG pipelines with LLM-Blender, we have used the MixInstruct (D. Jiang, Ren, and B. Y. Lin 2023) and BillSum (Kornilova and Eidelman 2019) datasets.

5.1 MixInstruct

MixInstruct is a benchmark dataset curated by the authors of LLM-Blender to benchmark ensemble models for LLMs in instruction-following tasks. It is a collection from a large-scale set of instruction examples primarily from four sources, Alpaca-GPT4, Dolly-15K, GPT4-ALL-LAION and ShareGPT. There are 100k examples for training, 5k for validation, and 5k for testing.

The dataset consists of:

- ID: A unique identifier for each input-instruction pair.
- Input: The input text or data that will be passed in the prompt.
- Instruction: The specific instruction or task to be performed, which will be provided along with the Input in the prompt.
- Output: The expected or desired output, which is the generated text that the model should aim to produce.

5.2 BillSum

The BillSum dataset is a corpus for summarization of US Legislation Bills. The corpus contains the text of bills and human-written summaries from the US Congress and California Legislature. The dataset is a corpus of 33,422 Bills. The dataset consists of:

- Text: The text present in the Congressional or State Bills.
- Summary: The hand-written summary of the Bills.
- Title: The title or name of the Congressional or State Bills.

6 LLMs used in RAG Pipelines

The authors of the LLM-Blender paper evaluated the performance of their framework on open-source LLMs like Open Assistant (LAION-AI 2023) , Vicuna (Chiang et al. 2023), Alpaca (Taori et al. 2023), Baize (Xu et al. 2023), MOSS (Sun and Qiu 2023), ChatGLM (Du et al. 2022), Koala (Geng et al. 2023), Dolly V2 (Conover et al. 2023), Mosaic MPT (MosaicML 2023), StableLM (Stability-AI 2023) and Flan-T5 (Chung et al. 2022). These LLMs were used to replicate the results presented in the original paper.

To further evaluate the performance of LLM-Blender, we consider the current top-ranking LLMs on popular benchmarks: the Open LLM Leaderboard (Beeching et al. 2023), LMSYS Leaderboard (Zheng et al. 2023), MMLU Benchmark (Massive Multi-task Language Understanding) (Hendrycks et al. 2020) and MT-Bench (Zheng et al. 2023).

The Open LLM Leaderboard runs the Eleuther AI LM Evaluation Harness (Gao et al. 2023) to evaluate the performance of LLMs on varied tasks. It uses benchmarks like ARC, HellaSwag, MMLU, TruthfulQA, Winogrande, and GSM8K to assess reasoning, commonsense, knowledge, truthfulness, and language understanding. The LMSYS Leaderboard provides an Elo rating, which serves as a predictor of win rates between models, helping to determine the comparative strengths of different LLMs. The MMLU Benchmark is a comprehensive evaluation designed to measure a text model's multitask accuracy by evaluating models in zero-shot and few-shot settings. The MT-Bench, on the other hand, measures the ability of LLMs to engage in coherent, informative, and engaging conversations. It is designed to assess the conversation flow and instruction-following capabilities of LLMs.

Based on rankings in these benchmarks, we selected the following LLMs for evaluating the pipelines with LLM-Blender: LLama-3-8B (Touvron et al. 2023), Phi-3-mini-128k-instruct (Abdin et al. 2024), Mistral-7B-Instruct-v0.2 (A. Q. Jiang et al. 2023), Starling-LM-7B-alpha (Zhu et al. 2023), SOLAR-10.7B-Instruct-v1.0 (D. Kim et al. 2023), OpenHermes-2.5-Mistral-7B (Teknium 2023) and OpenChat-3.5-0106 (G. Wang et al. 2023). These LLMs perform significantly better than the LLMs used by the authors in the original LLM-Blender paper, as evidenced by their superior rankings on the Open LLM Leaderboard, LMSYS Leaderboard, MMLU, and MT-Bench.

6.1 Mistral-Instruct-v0.2

The Mistral-7B-Instruct-v0.2 is a 7 billion parameter instruction-tuned language model based on the Mistral-7B architecture. It utilizes techniques like Grouped-Query Attention, Sliding-Window Attention, and a Byte-fallback BPE tokenizer. With a maximum context length of 8192 tokens, this model is designed for complex natural language tasks such as text understanding, transformation, and code generation.

The model has the following prompt format:

```
<s>[INST] {instruction} {prompt} [/INST]</s>
```

6.2 Starling-LM-7B-alpha

Starling-LM-7B-alpha is a 7 billion parameter open-source language model trained using Reinforcement Learning from AI Feedback (RLAIF) on the GPT-4 labeled Nectar dataset. It was initialized from OpenChat 3.5 and further optimized with the Starling-RM-7B-alpha reward model and Advantage-Induced Policy Alignment. The maximum context length is 8192 tokens.

The model has the following prompt format:

```
GPT4 Correct User: {instruction}
{prompt}<|end_of_turn|>GPT4 Correct Assistant:
```

6.3 SOLAR-10.7B

SOLAR-10.7B is a powerful 10.7 billion parameter language model demonstrating state-of-the-art performance in natural language tasks. Despite its compact size compared to models with over 30 billion parameters, it outperforms them by integrating weights from the Mistral 7B model into its upscaled layers. SOLAR-10.7B has a maximum context length of 8192 tokens.

The model has the following prompt format:

```
<s>### User: {instruction} {prompt}
### Assistant:
</s>
```

6.4 OpenHermes-2.5-Mistral-7B

OpenHermes-2.5-Mistral-7B is a 7 billion parameter Mistral-based language model fine-tuned on over 1 million GPT-4 generated examples and other high-quality code datasets. Built on the powerful Mistral architecture, it excels at text understanding, transformation, and code generation tasks. The maximum context length is 8192 tokens.

The model has the following prompt format:

```
<|im_start|>system
{instruction}<|im_end|>
<|im_start|>user
{prompt}<|im_end|>
<|im_start|>assistant
```

6.5 OpenChat-3.5

OpenChat-3.5 is a 7 billion parameter open-source language model fine-tuned using C-RLFT, an offline reinforcement learning strategy. Despite training on mixed-quality data without preference labels, it achieves performance on par with ChatGPT. OpenChat-3.5 has a maximum context length of 8192 tokens.

The model has the following prompt format:

```
GPT4 Correct User: {instruction}
{prompt}<|end_of_turn|>GPT4 Correct Assistant:
```

6.6 Phi-3-mini-128k-instruct

The Phi-3-Mini-128K-Instruct is a 3.8 billion-parameter, lightweight, state-of-the-art open model trained using the Phi-3 datasets. When assessed against benchmarks testing common sense, language understanding, math, code, long context and logical reasoning, Phi-3 Mini-4K-Instruct showcased a robust and state-of-the-art performance among models with less than 13 billion parameters. Phi-3-mini-128k-instruct has a maximum context length of 131072 tokens.

The model has the following prompt format:

```
<|user|>
{instruction} {prompt}<|end|>
<|assistant|>
```

6.7 Llama-3-8B

Llama-3-8B is an auto-regressive language model that uses an optimized transformer architecture. The tuned versions use supervised fine-tuning (SFT) and reinforcement learning with human feedback (RLHF) to align with human preferences for helpfulness and safety. Llama 3 uses a tokenizer with a vocabulary of 128K tokens that encodes language much more efficiently, which leads to substantially improved model performance. Llama3 has adopted grouped query attention (GQA) across both the 8B and 70B sizes. The model has a maximum context length of 8192 tokens.

The model has the following prompt format:

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>
{instruction}<|eot_id|><|start_header_id|>user<|end_header_id|>
{prompt}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

7 Metrics for evaluating RAG Pipelines

The authors of LLM-Blender evaluate the performance based on three natural language generation (NLG) evaluation metrics: BERTScore (Zhang et al. 2019), BLEURT (Sellam, Das, and Parikh 2020), and BARTScore (Yuan, Neubig, and P. Liu 2021). These NLG metrics have been used to evaluate the performance of the RAG pipelines with LLM-Blender.

7.1 BERTScore

The BERTScore is an automatic evaluation metric used for assessing the quality of text generated by natural language generation systems. Unlike existing popular methods that compute token-level syntactic similarity, BERTScore focuses on computing semantic similarity between tokens of the reference and hypothesis texts. A higher BERTScore indicates that the generated text is more semantically similar to the input text, which is desirable.

7.2 BLEURT

The BLEURT (BERT-based Learned Evaluation of User Responses with Transformers) is a metric used for evaluating text generation systems. It focuses on understanding the meaning of the generated text and comparing it to the expected output. BLEURT uses contextual embeddings from BERT to calculate similarity between words in the candidate and reference sentences. A higher BLEURT score, ranging from 0 to 1 or 1 to 100, represents higher similarity between the generated text and the reference text.

7.3 BARTScore

The BARTScore is an unsupervised metric that evaluates the semantic similarity between a reference text and a hypothesis text. It uses contextual token embeddings from BART to calculate similarity between words in the candidate and reference sentences. A higher BARTScore indicates that the hypothesis text is more semantically similar to the reference text, which is desirable.

8 Haystack Pipelines with LLM-Blender

A custom Haystack component, `LLMBlenderRanker`, has been implemented to integrate the LLM-Blender model for ranking answer candidates. `LLMBlenderRanker` implements the PairRanker module, which performs pairwise comparison of the input candidates and ranks them accordingly.

The component first loads the LLM-Blender model. It takes a list of generated answers as input. These answers are grouped based on their corresponding queries, and the PairRanker model is used to assign ranks to the answer candidates within each query group. The ranked answers are then structured into a list of GeneratedAnswer objects, where each object contains the input query and the ranked documents for that query. This list of GeneratedAnswer objects is returned as the output of the component.

The implementation of the `LLMBlenderRanker` component involves the following key methods:

- `generate_inputs_candidates` method: This method processes the list of answers and organizes them based on their corresponding query inputs. It creates lists of candidates for each query. If the length of the candidate list is less than the length of the smallest candidate list among all queries, the candidate list is trimmed to match the length of the smallest candidate list. It returns the grouped inputs and candidates.

```
def _generate_inputs_candidates(
    self,
    answers_list: List[List[GeneratedAnswer]],
) -> Tuple[List[str], List[List[str]], List[List[Dict[str, Any]]]]:
    """
    Generate candidates for each query by combining all answers where the
    query (input) is the same.

    If the length of the candidate list is less than the length of the
    smallest candidate list among all queries, the candidate list is
    trimmed to match the length of the smallest candidate list.

    :param answers_list:
```

```
        A list of lists of answers.
    :return:
        A list of inputs, a list of lists of candidates, and a list of
    lists of metadata.
    """

    inputs_candidates_meta = defaultdict(list)
    for answers in answers_list:
        for generated_answer in answers:
            inputs_candidates_meta[generated_answer.query].append((
    generated_answer.data, generated_answer.meta))

    # Find the smallest length among all candidate lists for each query
    lengths = {query: len(candidates_list) for query, candidates_list in
    inputs_candidates_meta.items()}
    min_length = min(lengths.values())

    # Trim each candidate list to match the smallest length
    for query, candidates_list in inputs_candidates_meta.items():
        inputs_candidates_meta[query] = list(candidates_list[:min_length])

    inputs = list(inputs_candidates_meta.keys())
    candidates_meta = list(inputs_candidates_meta.values())
    candidates = [[data for data, _ in lst] for lst in candidates_meta]
    meta = [[meta for _, meta in lst] for lst in candidates_meta]

    return inputs, candidates, meta
```

Listing 1: The custom `LLMBlenderRanker` Haystack component.

- `generate_answers_ranked` method: This method takes the inputs, candidates, and ranks and generates a list of GeneratedAnswer objects based on the ranked candidates for each input. It first creates a dictionary where the keys are the input queries, and the values are lists of tuples containing the candidate answers and their corresponding ranks. This dictionary is then sorted based on the ranks, and the sorted candidates are extracted to create the GeneratedAnswer objects.

```
def _generate_answers_ranked_candidates(
    self,
    inputs: List[str],
    candidates: List[List[str]],
    ranks_list: List[List[int]],
    meta: List[List[Dict[str, str]]],
) -> List[GeneratedAnswer]:
    """
    Generate the ranked candidates for each input using the ranks from the
    Pair Ranker model.

    :param inputs:
        A list of inputs.
    :param candidates:
        A list of lists of candidates.
    :param ranks_list:
        A list of lists of ranks.
    :param meta:
        A list of lists of metadata.
    :return:
         A list of Generated Answers.
    """
    # Create a dictionary to store the ranked candidates for each input
    ranked_candidates = {}

    # Iterate through the inputs and ranks
    for i in range(len(inputs)):
        input_str = inputs[i]
        ranks = ranks_list[i]
        candidates_for_input = candidates[i]
        meta_for_input = meta[i]

        # Store the candidates, their ranks, and their metadata in a
    dictionary
        ranked_candidates[input_str] = list(zip(candidates_for_input, ranks
    , meta_for_input))
```

```
    # Sort the dictionary based on the ranks and extract the sorted
    candidates
    sorted_candidates = {key: sorted(values, key=lambda item: item[1]) for
    key, values in ranked_candidates.items()}

    # Convert the sorted candidates to a list of Generated Answers for each
     input
    ranked_generated_answers = [
        [
            GeneratedAnswer(query=input_str, data=candidate, documents=[],
    meta=meta)
            for candidate, _, meta in sorted_candidates[input_str]
        ]
        for input_str in inputs
    ]

    ranked_generated_answers = list(chain.from_iterable(
    ranked_generated_answers))

    return ranked_generated_answers
```

Listing 2: The custom `LLMBlenderRanker` Haystack component.

- `run` method: This method is responsible for ranking the output answers using the PairRanker model. It calls the `generate_inputs_candidates` method to group the inputs and candidates, and then uses the PairRanker model to obtain the ranks for each group. Finally, it calls the `generate_answers_ranked` method to create the list of GeneratedAnswer objects with the ranked answers, which is returned as the output.

```
@component.output_types(documents=List[GeneratedAnswer])
def run(self, answers: Variadic[List[GeneratedAnswer]]):
    """
    Rank the output answers using the LLM Blender model.

    :param answers:
        A list of answers to be ranked.
    :return:
        A list of ranked answers.
    """

    if not answers:
        return {"answers": []}

    if self.model is None:
        msg = "The component LLMBlenderRanker wasn't warmed up. Run '
    warm_up()' before calling 'run()'."
        raise ComponentError(msg)

    inputs, candidates, meta = self._generate_inputs_candidates(answers)
    ranks = self.model.rank(inputs, candidates)
    ranked_answers = self._generate_answers_ranked_candidates(inputs,
    candidates, ranks, meta)

    return {"answers": ranked_answers}
```

Listing 3: The custom `LLMBlenderRanker` Haystack component.

To ensemble multiple RAG pipelines, we connect them to an LLMBlenderRanker. The LLMBlenderRanker performs pairwise comparisons between the output candidates from each pipeline for a given query. It ranks these output candidates based on their similarity or relevance to the query, using the LLM-Blender model. The ranked output candidates from the LLMBlenderRanker are then passed to the LLMBlenderFuser. The LLMBlenderFuser summarizes and combines the top-ranked outputs to generate a single fused output. This fused output leverages the strengths of different LLMs and their respective pipelines, producing a robust and comprehensive answer by combining the best outputs for a given query.

The code for custom `LLMBlenderRanker` component can be found at https://github.com/avnlp/llmsas-rankers/blob/main/src/llms_as_rankers/llm_blender

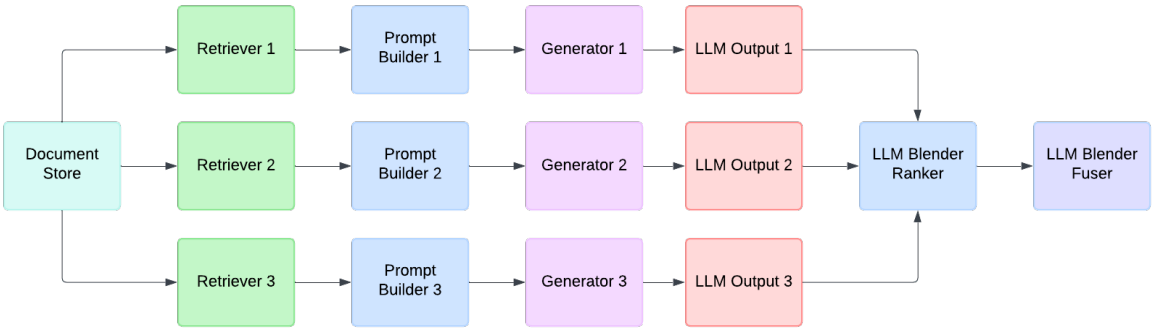

Figure 3: Ensembling RAG pipelines using the LLM-Blender Ranker and Fuser.

By ensembling multiple RAG pipelines through this process, the system can harness the benefits of different LLMs and retrieval strategies. During each pipeline run, the top-performing outputs from each pipeline are selected and fused, resulting in more accurate and reliable answers compared to relying on a single pipeline or LLM.

9 Experiments

To assess the the effectiveness of the PairRanker component, a series of experiments were conducted across various datasets. These experiments were divided into three distinct cases, each focusing on a specific aspect of the pipeline, to correctly measure the effectiveness of the PairRanker when used with different LLMs. The outputs generated in these cases were evaluated using the BERTScore, BLEURT, and BARTScore metrics.

- **Case 1:** Establishing Baselines with Individual LLMs

  In the first case, each LLM was used independently to generate outputs for each dataset. These outputs were then evaluated using the three metrics (BERTScore, BLEURT, and BARTScore). This step established baseline performance levels for the individual LLMs, providing a reference point for comparison with the ensemble PairRanker approach.

Figure 4: Pipeline with a LLM

- **Case 2:** Evaluating the PairRanker with all available LLMs

  The second case involved creating a pipeline that combined all available LLMs with the PairRanker component. For each example in the dataset, every LLM generated an output based on the provided inputs and instructions. The PairRanker then performed pairwise comparisons and rankings of these outputs, selecting the top-ranked output as the final result. The pipeline's performance was evaluated using the BERTScore, BLEURT, and BARTScore metrics. This measured the effectiveness of the PairRanker in selecting the most relevant and high-quality output from the diverse set of LLM-generated candidates.

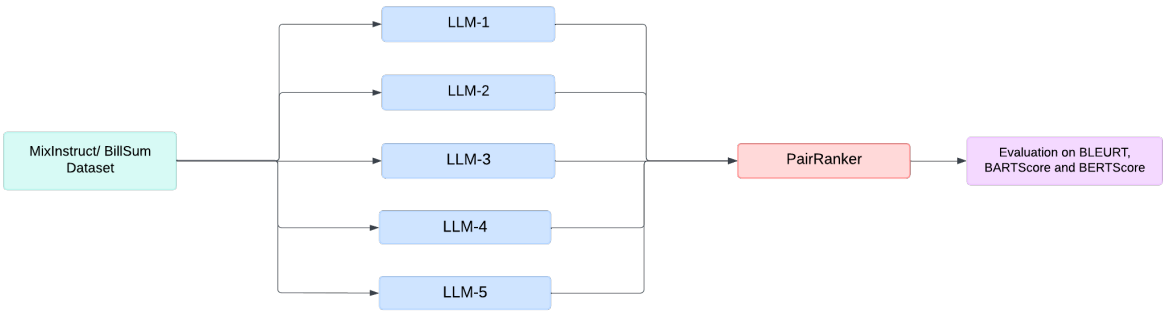

Figure 5: Pipeline with 5 LLMs and a PairRanker

- **Case 3:** Ensembling Top-Performing LLMs with the PairRanker

  In the third case, the top three language models (LLMs) that performed best in the initial evaluation (Case 1), as measured by the BERTScore, BLEURT, and BARTScore metrics, were ensembled using the PairRanker. A pipeline was creating using top 3 performing LLMs and the PairRanker component. Similar to Case 2, for each example, the PairRanker ranked the outputs from the three LLMs, and the top-ranked output was selected for evaluation. The pipeline's performance was evaluated using the BERTScore, BLEURT, and BARTScore metrics. This approach took advantage of the collective strengths of the best-performing LLMs while utilizing the PairRanker's ability to select the most relevant and high-quality output.

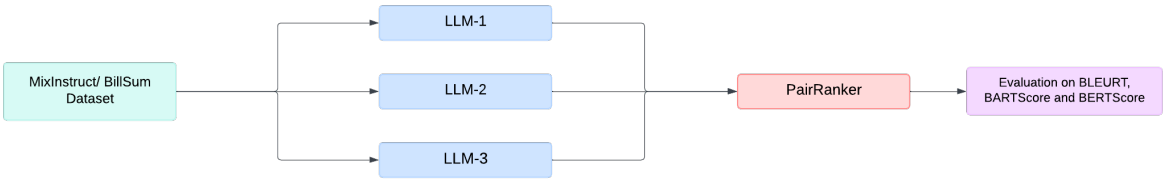

Figure 6: Pipeline with top performing 3 LLMs and a PairRanker

Haystack pipelines were created for all the three cases for both the MixInstruct and BillSum datasets. For both the datasets, the LLMs: Llama-3-8B, Phi-3-mini-128k-instruct, Mistral-7B-Instruct-v0.2, Starling-LM-7B-alpha, SOLAR-10.7B-Instruct-v1.0, OpenHermes-2.5-Mistral-7B and OpenChat-3.5-0106 were used. The PairRM, PairRanker model was used. The pipelines were evaluated on the BARTScore, BLEURT and BERTScore metrics. In the evaluation, we give higher preference to BARTScore since the authors of the LLM-Blender found that BARTScore shows higher correlation with GPT-Rank and other NLG metrics.

9.1 Experiments on the MixInstruct Dataset

We replicate the results presented in the LLM-Blender paper by evaluating the performance of Open Assistant, Vicuna, Alpaca, Baize, MOSS, ChatGLM, Koala, Dolly V2, Mosaic MPT, StableLM and Flan-T5 with the PairRanker on the MixInstruct dataset.

We also evaluated five LLMs: Llama-3-8B, Phi-3-mini-128k-instruct, Mistral-7B-Instruct-v0.2, Starling-LM-7B-alpha, SOLAR-10.7B-Instruct-v1.0, OpenHermes-2.5-Mistral-7B and OpenChat-3.5-0106, each paired with the PairRanker on the MixInstruct dataset. Each pipeline's performance was evaluated using the BARTScore, BLEURT, and BERTScore metrics.

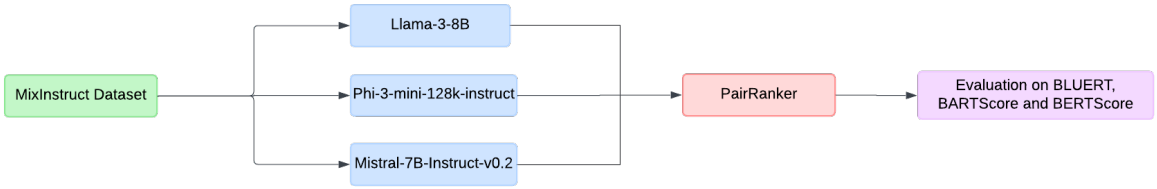

Figure 7: Pipeline with three LLMs (Llama-3-8B, Phi-3-mini-128k-instruct and Mistral-7B-Instruct-v0.2) and a PairRanker on the MixInstruct dataset.

The Llama-3-8B, Phi-3-mini-128k-instruct, and OpenChat-3.5-0106 were the top performing LLMs based on the three metrics. The PairRanker combined with Llama-3-8B, Phi-3-mini-128k-instruct and Mistral-7B-Instruct-v0.2 gave the highest scores on the MixInstruct dataset.

Model	**BERTScore**	**BARTScore**	**BLEURT**
PairRM - Llama-3, Phi-3 and Mistral-Instruct-v0.2	75.83	**-2.87**	-0.26
Llama-3	**76.86**	-2.90	-0.27
Phi-3-Mini-128K-Instruct	74.32	-2.94	-0.28
OpenChat-3.5-0106	75.14	-3.02	-0.25
OpenHermes-2.5-Mistral-7B	76.05	-3.03	**-0.23**
SOLAR-10.7B-Instruct-v1.0	73.57	-3.06	-0.33
Starling-LM-7B-alpha	73.49	-3.07	-0.32
Mistral-7B-Instruct-v0.2	72.62	-3.17	-0.41

Table 1: Evaluation Scores on the MixInstruct Dataset

9.2 Experiments on the BillSum Dataset

The PairRanker's efficacy was evaluated on the BillSum dataset when paired with LLMs: Llama-3-8B, Phi-3-mini-128k-instruct, Mistral-7B-Instruct-v0.2, Starling-LM-7B-alpha, SOLAR-10.7B-Instruct-v1.0, OpenHermes-2.5-Mistral-7B and OpenChat-3.5-0106. The pipelines were evaluated on the BARTScore, BLEURT and BERTScore metrics.

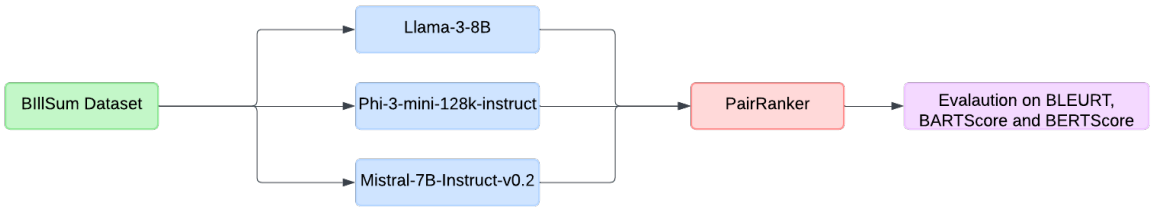

Figure 8: Pipeline with three LLMs (Llama-3-8B, Phi-3-mini-128k-instruct and Mistral-7B-Instruct-v0.2) and a PairRanker on the BillSum dataset.

The Llama-3-8B, SOLAR-10.7B-Instruct-v1.0 and OpenHermes-2.5-Mistral-7B were the top performing LLMs based on the three metrics. The PairRanker combined with Llama-3-8B, Phi-3-mini-128k-instruct and Mistral-7B-Instruct-v0.2 gave the highest scores on the BillSum dataset.

Model	BERTScore	BARTScore	BLEURT
PairRM - Llama-3, Phi-3 and Mistral-Instruct-v0.2	**75.43**	**-3.19**	**-0.20**
Llama-3	74.67	-3.23	-0.25
SOLAR-10.7B-Instruct-v1.0	73.27	-3.24	-0.30
OpenHermes-2.5-Mistral-7B	74.55	-3.41	-0.24
Phi-3-Mini-128K-Instruct	74.52	-3.42	-0.26
OpenChat-3.5-0106	75.35	-3.44	-0.21
Mistral-7B-Instruct-v0.2	73.91	-3.48	-0.39
Starling-LM-7B-alpha	74.66	-3.49	-0.23

Table 2: Evaluation Scores on the BillSum Dataset

10 Results

- A custom component, *LLMBlenderPairRanker*, was developed to integrate the LLM-Blender Framework with Haystack Pipelines. Haystack RAG Pipelines with the LLM-Blender component to ensemble LLMs were evaluated. The pipelines were evaluated on the BillSum and MixInstruct datasets using three metrics: BARTScore, BLEURT, and BERTScore.

- We successfully replicated the previously reported results for the LLM-Blender. Moreover, significantly improved performance was observed when utilizing newer LLM models, such as Llama-3-8B, Phi-3-mini and Mistral-7B. These findings demonstrate the potential of ensembling state-of-the-art LLMs to enhance the performance of RAG Pipelines on question-answering, summarization and instruction-following tasks.

- The authors of LLM-Blender obtained BERTScore values in the range of 62.26 to 74.68 on the MixInstruct dataset. They obtained a BERTScore value of 72.97 with the PairRanker. We obtained BERTScore values in the range of 72.62 to 76.86 using the newer LLMs. We obtained a BERTScore value of 75.83 with the PairRanker ensembling the results from Llama-3-8B, Phi-3-mini and Mistral-7B.

- The authors of LLM-Blender obtained BARTScore values in the range of -4.57 to -3.14 on the MixInstruct dataset. They obtained a BARTScore value of -3.14 with the PairRanker. We obtained BARTScore values in the range of -3.17 to -2.87 using the newer LLMs. We obtained a BARTScore value of -2.87 with the PairRanker ensembling the results from Llama-3-8B, Phi-3-mini and Mistral-7B.

- The authors of LLM-Blender obtained BLEURT values in the range of -1.23 to -0.37 on the MixInstruct dataset. They obtained a BLEURT value of -0.37 with the PairRanker. We obtained BLEURT values in the range of -0.41 to -0.23 using the newer LLMs. We obtained a BLEURT value of -0.26 with the PairRanker ensembling the results from Llama-3-8B, Phi-3-mini and Mistral-7B.

- The newer models like Llama-3-8B, Phi-3-mini, and Mistral-7B significantly outperformed all the models used by the LLM Blender authors on all the three metrics: BERTScore, BARTScore and BLEURT on the MixInstruct dataset.

- On the BillSum dataset, we obtained BERTScore values from 73.91 to 75.43, BARTScore values from -3.49 to -3.19, and BLEURT values from -0.39 to -0.20 across the different LLMs. The PairRanker model, ensembling the outputs from Llama-3-8B, Phi-3-mini, and Mistral-7B, achieved the highest scores of 75.83 for BERTScore, -3.19 for BARTScore, and -0.20 for BLEURT.

- For both the BillSum and MixInstruct datasets, the PairRanker model achieved the best performance when ensembling the outputs from Llama-3-8B, Phi-3-mini, and Mistral-7B. This combination of LLMs, ensembled using the LLM Blender, significantly outperformed each individual model's performance on all the evaluation metrics.

References

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. (2023). "Mistral 7B". In: *arXiv preprint arXiv:2310.06825*. URL: https://arxiv.org/abs/2310.06825.

Anastassia Kornilova and Vlad Eidelman (2019). "BillSum: A corpus for automatic summarization of US legislation". In: *arXiv preprint arXiv:1910.00523*. URL: https://arxiv.org/abs/1910.00523.

Banghua Zhu, Evan Frick, Tianhao Wu, Hanlin Zhu, and Jiantao Jiao (2023). "Starling-7B: Improving LLM Helpfulness and Harmlessness with RLAIF". In: *HuggingFace*. URL: https://huggingface.co/berkeley-nest/Starling-LM-7B-alpha.

Canwen Xu, Daya Guo, Nan Duan, and Julian McAuley (2023). "Baize: An open-source chat model with parameter-efficient tuning on self-chat data". In: *arXiv preprint arXiv:2304.01196*. URL: https://arxiv.org/abs/2304.01196.

Dahyun Kim, Chanjun Park, Sanghoon Kim, Wonsung Lee, Wonho Song, Yunsu Kim, Hyeonwoo Kim, Yungi Kim, Hyeonju Lee, Jihoo Kim, et al. (2023). "Solar 10.7 b: Scaling large language models with simple yet effective depth up-scaling". In: *arXiv preprint arXiv:2312.15166*. URL: https://arxiv.org/abs/2312.15166.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt (2020). "Measuring Massive Multitask Language Understanding". In: *arXiv preprint arXiv:2009.03300*. URL: https://arxiv.org/abs/2009.03300.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin (2023). "LLM-Blender: Ensembling Large Language Models with Pairwise Ranking and Generative Fusion". In: *arXiv preprint arXiv:2306.02561*. URL: https://arxiv.org/abs/2306.02561.

Edward Beeching, Clémentine Fourrier, Nathan Habib, Sheon Han, Nathan Lambert, Nazneen Rajani, Omar Sanseviero, Lewis Tunstall, and Thomas Wolf (2023). "Open LLM Leaderboard". In: *HuggingFace*. URL: https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard.

Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu (2023). "Openchat: Advancing open-source language models with mixed-quality data". In: *arXiv preprint arXiv:2309.11235*. URL: https://arxiv.org/abs/2309.11235.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. (2023). "Llama 2: Open foundation and fine-tuned chat models". In: *arXiv preprint arXiv:2307.09288*. URL: https://arxiv.org/abs/2307.09288.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. (2022). "Scaling instruction-finetuned language models". In: *arXiv preprint arXiv:2210.11416*. URL: https://arxiv.org/abs/2210.11416.

Julian Salazar, Davis Liang, Toan Q Nguyen, and Katrin Kirchhoff (2019). "Masked Language Model Scoring". In: *arXiv preprint arXiv:1910.14659*. URL: https://arxiv.org/abs/1910.14659.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou (2024). "Mixture-of-Agents Enhances Large Language Model Capabilities". In: *arXiv preprint arXiv:2406.04692*. URL: https://arxiv.org/abs/2406.04692.

LAION-AI (2023). "Open Assistant". In: *https://github.com/LAION-AI/Open-Assistant*.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou (2023). "A framework for few-shot language model evaluation". In: *Zenodo 10.5281/zenodo.10256836*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. (2023). "Judging LLM-as-a-judge with MT-Bench and Chatbot Arena". In: *arXiv preprint arXiv:2306.05685*. URL: https://arxiv.org/abs/2306.05685.

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. (2024). "Phi-3 technical report: A highly capable language model locally on your phone". In: *arXiv preprint arXiv:2404.14219*. URL: https://arxiv.org/abs/2404.14219.

Mathieu Ravaut, Shafiq Joty, and Nancy F Chen (2022). "SummaReranker: A multi-task mixture-of-experts re-ranking framework for abstractive summarization". In: *arXiv preprint arXiv:2203.06569*. URL: https://arxiv.org/abs/2203.06569.

Mike Conover, Matt Hayes, Ankit Mathur, Xiangrui Meng, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin (2023). "Free Dolly: Introducing the World's First Truly Open Instruction-Tuned LLM". In: *https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm*.

NLP Team MosaicML (2023). "Introducing MPT-7B: A New Standard for Open-Source, ly Usable LLMs". In: *https://www.mosaicml.com/blog/mpt-7b*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. (2020). "Retrieval-augmented generation for knowledge-intensive nlp tasks". In: *Advances in Neural Information Processing Systems*. URL: https://arxiv.org/abs/2005.11401.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto (2023). "Stanford Alpaca: An Instruction-following LLaMA model". In: *https://github.com/tatsu-lab/stanford*$_a$*lpaca*.

Stability-AI (2023). "StableLM: Stability AI Language Models". In: *https://github.com/stability-AI/stableLM*.

Teknium (2023). "OpenHermes 2.5: An Open Dataset of Synthetic Data for Generalist LLM Assistants". In: *HuggingFace*. URL: https://huggingface.co/datasets/teknium/OpenHermes-2.5.

Thibault Sellam, Dipanjan Das, and Ankur P Parikh (2020). "BLEURT: Learning robust metrics for text generation". In: *arXiv preprint arXiv:2004.04696*. URL: https://arxiv.org/abs/2004.04696.

Tianxiang Sun and Xipeng Qiu (2023). "MOSS". In: *https://github.com/OpenLMLab/MOSS*.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi (2019). "Bertscore: Evaluating text generation with bert". In: *arXiv preprint arXiv:1904.09675*. URL: https://arxiv.org/abs/1904.09675.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing (2023). "Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality". In: URL: https://lmsys.org/blog/2023-03-30-vicuna/.

Weizhe Yuan, Graham Neubig, and Pengfei Liu (2021). "Bartscore: Evaluating generated text as text generation". In: *Advances in Neural Information Processing Systems*. URL: https://arxiv.org/abs/2106.11520.

Xinyang Geng, Arnav Gudibande, Hao Liu, Eric Wallace, Pieter Abbeel, Sergey Levine, and Dawn Song (2023). "Koala: A Dialogue Model for Academic Research". In: *https://bair.berkeley.edu/blog/2023/04/03/koala/*.

Yixin Liu and Pengfei Liu (2021). "SimCLS: A simple framework for contrastive learning of abstractive summarization". In: *arXiv preprint arXiv:2106.01890*. URL: https://arxiv.org/abs/2106.01890.

Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang (2022). "GLM: General language model pretraining with autoregressive blank infilling". In: *arXiv preprint arXiv:2103.10360*. URL: https://arxiv.org/abs/2103.10360.