Title

Marco Coppola,
Valerio Pio De Nicola,
Davide De Rosa

University of Bologna

1 Introduction

per la fine

2 InstructLab

2.1 Introduction to LAB

Large language models (LLMs) have achieved success in NLP tasks due to the transformer architecture. LLM training consists of two main phases: **pre-training** (which is computationally expensive and involves predicting the next token using massive datasets) and **alignment tuning** (which refines model behavior through instruction and preference tuning).

Pre-training dominates the resource cost, while alignment tuning, including instruction tuning (training on task-specific instructions) and preference tuning (using human feedback methods like RLHF), requires significantly fewer data and compute resources. Despite this, alignment tuning is crucial for optimizing LLMs for real-world use.

To address challenges in scaling alignment tuning, the **LAB** (**Large-scale Alignment for chatBots**[1]) **method** has been recently created. LAB includes:

1. **Synthetic data generation** guided by taxonomy and quality assurance, avoiding reliance on proprietary LLMs or extensive human curation.
2. **A novel multi-phase training framework** that integrates new knowledge without causing catastrophic forgetting.

LAB-trained models achieve competitive performance compared to those using human-annotated or GPT-4-generated data, demonstrating its effectiveness for improving LLM instruction-following capabilities.

Recent advancements in instruction tuning for large language models (LLMs) have primarily relied on two key methodologies: **human-annotated datasets** and **synthetic data generation**. Traditional approaches, such as those pioneered by OpenAI and later adopted by the creators of LLaMA 2, emphasize the collection of high-quality human-generated data. This process involves extensive human annotation efforts, requiring rigorous selection and training of annotators to ensure consistency and quality.

While effective in aligning models with human preferences, these methods are resource-intensive, costly, and often slow, limiting the ability to rapidly explore new instruction types and model capabilities. To address these limitations, synthetic data generation has emerged as an alternative, leveraging LLMs to produce instruction-tuning datasets.

Early efforts, such as Self-Instruct, introduced a bootstrapping approach that expands a small set of human-written seed instructions into a large dataset using an LLM's own generation capabilities. Subsequent refinements sought to enhance data diversity and instruction complexity through iterative and principled augmentation techniques.
Other works have further extended this approach by focusing on task diversity and progressive training frameworks, incorporating richer reasoning signals to improve model performance incrementally.

More recently, semi-automated techniques, such as GLAN, have introduced a structured approach to synthetic data generation by leveraging human-curated taxonomies.
However, such methods remain constrained by the limitations of the teacher model used for data generation, particularly when relying on proprietary models like GPT-4, which impose restrictions on the commercial usability of the generated data.

In contrast, open-source approaches, such as LAB, attempt to mitigate these concerns by employing models like Mixtral, allowing for greater flexibility in data generation and application.

The ongoing evolution of instruction tuning methods highlights the trade-offs between human-annotated and synthetic data-driven approaches.
While human-generated data ensures high-quality alignment, it is costly and inflexible.
Conversely, synthetic data techniques offer scalability and efficiency but introduce challenges related to data diversity, quality control, and legal constraints surrounding proprietary models.
Future research will likely focus on hybrid approaches that balance these trade-offs, incorporating the strengths of both methodologies to enhance the instruction-tuning process.

2.2 How does LAB work

LAB is a structured framework designed to enhance instruction tuning through a combination of systematic data curation and multi-phased training. It consists of two key components:

- **Taxonomy-Guided Synthetic Data Generator**: this component facilitates data curation by organizing instructions into a structured taxonomy. It also plays a crucial role in guiding the synthetic data generation process, ensuring both high diversity and quality in the instruction-tuning dataset. By leveraging a well-defined taxonomy, LAB aims to improve the comprehensiveness and relevance of the generated data.

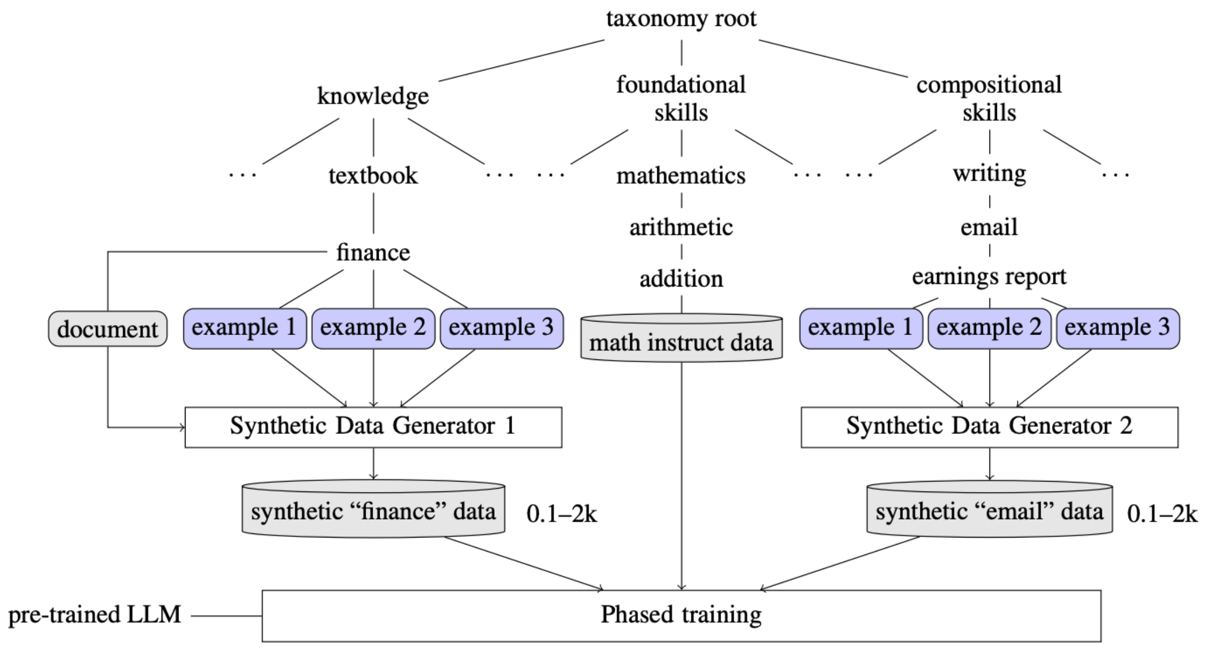


Figure 1: *Overview of the LAB alignment method. Starting from the taxonomy root, data are curated in each top-level groups and examples in the leaf nodes are used by the synthetic data generators to generate orders of magnitude data for the phased-training step for instruct-tuning.*

- **Multi-Phased Instruction-Tuning with Replay Buffers**: to maintain training stability and mitigate issues such as catastrophic forgetting, LAB employs a multi-stage tuning approach. The use of replay buffers allows the model to retain previously learned information while adapting to new instructions, thereby improving overall alignment performance.

Together, these components form an end-to-end pipeline for aligning a pre-trained LLM, as illustrated in Figure 1. This approach balances synthetic data diversity with training stability, making it a scalable and effective method for large-scale instruction tuning.

2.2.1 Taxonmy

LAB's taxonomy serves as a structured framework for organizing instruction-tuning data, enabling systematic curation and enhancement of training datasets. It hierarchically classifies data samples into three main branches: **knowledge**, **foundational skills**, and **compositional skills**. Each of these branches is further subdivided into more granular levels, with specific tasks defined at the leaf nodes. These leaf nodes are exemplified by manually written instruction-response pairs, ensuring clarity in task representation.

This hierarchical organization provides several advantages.

First, it allows model designers and data curators to identify gaps in the model's capabilities by pinpointing missing or underrepresented tasks.

Second, it facilitates the incremental expansion of training data, as new tasks can be seamlessly integrated by adding a leaf node under the appropriate branch, accompanied by 1–3 examples.

This structured approach ensures that instruction tuning remains both comprehensive and adaptable, improving the overall alignment and robustness of the trained LLM.

Knowledge
The *Knowledge* branch categorizes domain-specific documents like textbooks and research papers. It ensures ethical synthetic data generation by selecting only licensed content.

Foundational Skills
The *Foundational Skills* branch includes core abilities like math, coding, and reasoning. Public datasets support structured learning for effective generalization.

Compositional Skills
The *Compositional Skills* branch integrates knowledge and foundational skills for complex tasks. It enables models to synthesize information for coherent, multi-step responses.

2.2.2 Taxonomy-Driven Synthetic Data Generator

LAB enhances **synthetic data generation** (**SDG**) by leveraging a *taxonomy-driven approach* rather than relying on traditional random sampling methods.
While manually curated data samples embedded in the taxonomy's leaf nodes can be used for direct instruction tuning, prior research suggests that a large volume of high-quality instruction data is necessary for improving instruction-following capabilities in LLMs.

Existing SDG methods attempt to scale synthetic data generation using teacher models, but they suffer from mode collapse — over-representing the dominant patterns of the teacher model while neglecting diverse or less common instruction types.

This limitation arises from random selection of seed examples, which results in prompts that reflect an "average" of the dataset rather than task-specific examples. Consequently, the teacher model generates synthetic data that predominantly aligns with its dominant distribution while ignoring the long-tail of diverse or nuanced tasks. To overcome this, LAB replaces random selection with taxonomy-driven sampling, ensuring that data generation is targeted at the level of individual leaf nodes. This approach guarantees balanced representation across different instruction types, preventing bias toward a subset of commonly occurring prompts.

LAB introduces two new SDG methods based on this taxonomy-guided framework:

1. **Skills Generation**: this method uses the task examples stored in the leaf nodes to generate a larger dataset using the open-source **Mixtral-7x8B model**. By focusing on specific skill categories, LAB ensures better diversity and task-specific representation in the synthetic data.
2. **Knowledge Generation**: unlike prior approaches that rely on a teacher model's internal knowledge, LAB's knowledge generation method still employs **Mixtral-7x8B**, but without depending on pre-existing knowledge stored in the model. This

approach mitigates biases introduced by the teacher model's knowledge limitations while maintaining control over the source material used for synthetic data creation.

By structuring data generation around the taxonomy, **LAB improves both the diversity and quality** of synthetic instruction-tuning datasets, addressing key weaknesses in traditional SDG pipelines.

Skill Generation
Skills-SDG follows a structured, multi-stage approach to generate diverse, high-quality instructional data using Mixtral-7x8B. Four specialized prompts guide the teacher model in different roles:

1. **Instruction Generation**: the model generates diverse, well-structured instructions by systematically exploring taxonomy nodes.

2. **Evaluating Instructions**: instructions are filtered for relevance, safety, and feasibility, ensuring only high-quality queries proceed.

3. **Generating Responses**: responses are tailored per domain, ensuring creativity for writing tasks and precision for STEM subjects.

4. **Evaluating Instruction-Response Pairs**: a rating system filters out inaccurate or off-topic samples, ensuring high-quality data.

Knowledge Generation
Knowledge-SDG addresses SDG limitations by grounding data generation in external sources like manuals and books, reducing hallucinations. Unlike conventional SDG, this method ensures factual accuracy by integrating structured examples with external knowledge. The teacher model evaluates content to maintain reliability, making this approach particularly valuable for specialized domains.

2.3 What's under the hood

Fine-tuning large language models (LLMs) on consumer hardware presents significant challenges due to high computational and memory demands. To address these challenges, we leverage the ILAB framework, which integrates several optimization techniques: Synthetic Data Generation (SDG), Model Quantization, Low-Rank Adaptation (LoRA[2]), and Quantized LoRA (QLoRA[3]). These methods work together to enable efficient training and adaptation of LLMs on consumer-grade GPUs.

2.3.1 Synthetic Data Generation (SDG)

SDG enhances the training process by generating high-quality synthetic data, reducing the dependence on large real-world datasets. This not only helps in domain adaptation but also mitigates data scarcity issues, improving model generalization.

2.3.2 Model Quantization

Model quantization reduces the precision of model parameters (e.g., from 16-bit floating point to 8-bit integers), significantly lowering memory requirements and computational costs. This enables larger models to fit within the limited VRAM of consumer GPUs while maintaining near-original performance.

2.3.3 Low-Rank Adaptation (LoRA)

LoRA fine-tunes LLMs efficiently by introducing low-rank matrices into selected layers instead of updating all model parameters. This dramatically reduces the number of trainable parameters, making fine-tuning feasible on constrained hardware.

2.3.4 Quantized LoRA (QLoRA)

QLoRA combines the benefits of quantization and LoRA by applying quantization techniques to the base model while training LoRA adapters on top. This further reduces memory consumption, allowing large-scale models to be fine-tuned even on GPUs with limited VRAM.

2.3.5 How These Techniques Work Together

By combining these methods, ILAB creates a highly efficient fine-tuning pipeline.

- SDG provides a diverse training dataset, enhancing model robustness
- Model quantization compresses the base model, reducing memory footprint
- LoRA then enables efficient fine-tuning without modifying the entire model, and
- QLoRA further optimizes this by applying quantization to the base model while keeping adapter layers in higher precision.

This synergy allows consumer hardware, such as NVIDIA RTX-series GPUs, to fine-tune large-scale LLMs that would otherwise require enterprise-level infrastructure.

2.4 Output Model: GGUF Format with Llama CPP Runner

After fine-tuning the large language model (LLM) using the ILAB framework, the resulting model is saved in the GGUF (GPT-Generated Unified Format) format.
GGUF is a highly efficient format designed for easy deployment and fast inference, ensuring that the model can be loaded and run with minimal overhead on a variety of hardware configurations. This format is particularly well-suited for use with the Llama CPP runner, which is a lightweight inference engine designed for high-performance model execution.
One of the key advantages of the GGUF format, when paired with the Llama CPP runner, is its seamless compatibility with Apple's hardware ecosystem. Apple Silicon (M1, M2, and future generations) is considered a first-class citizen in this setup, benefiting from optimized performance through the use of several native technologies, including ARM NEON, Accelerate, and Metal frameworks.
This optimized execution on Apple Silicon enables efficient use of consumer hardware, making it possible to run the fine-tuned model with minimal latency and power consumption, even on relatively modest devices.

LAB enables practical fine-tuning of LLMs on consumer hardware by leveraging SDG, model quantization, LoRA, and QLoRA. These techniques collectively reduce memory requirements, computational overhead, and data dependency, making LLM fine-tuning and inference accessible beyond high-performance data centers.

3 Our Case Study

Our main goal was to fine-tune a Large Language Model (LLM) using consumer hardware. We chose to focus on creating a model which could help us write Jolie[1] code. We started by looking for tools that would help us achieve this task. We found IBM's InstructLAB[2], which is still in the early stages of development.

After an initial phase of studying the tool's documentation, we began by installing the tool.

3.1 Installing ILAB

ILAB is essentially a Python package, so it can be installed using pip in your virtual environment with the following command:

```
pip install instructlab
```

During the installation process, you need to set various parameters to enable GPU acceleration and other features. You can set up all the necessary parameters in your environment by running the following command:

```
ilab config init
```

Our main machine for this project was an *M1 Pro MacBook Pro 14*, with 16GB of RAM. We also used our department's HPC for various tests before settling on the MacBook.

After installing and setting everything up, we created the taxonomy.

3.2 Creating the Taxonomy

ILAB's taxonomy consists of different "leaves" called QnA (Questions and Answers) in YAML format. Each leaf covers a single topic.
These files are quite simple. An example of our Introduction to Jolie QnA is shown below:

[1]https://www.jolie-lang.org/
[2]https://instructlab.ai/

```yaml
version: 2
task_description: |
    Provide an understanding of the Jolie programming language, its features,
    and usage for service-oriented programming.
created_by: murkrow
seed_examples:
  - question: What is Jolie designed for?
    answer: Jolie is designed for service-oriented programming, focusing on APIs,
    concurrency, communication, and computation in microservice development.
  - question: How can I run a Jolie service saved in a file?
    answer: Save the Jolie code in a file, for example, `greeter.ol`, and run it
    from the terminal using the command `jolie greeter.ol`.
  - question: How does Jolie handle data and interoperability?
    answer: Jolie structures all data as trees, which can be semi-automatically or
    fully automatically converted between formats like JSON and XML, and communicated
    over protocols such as HTTP or binary protocols.
  - context: |
      Jolie provides primitives to deal directly with common concerns regarding
      microservices programming without relying on frameworks or external libraries.
    question: Why is Jolie considered the first language for microservices?
    answer: Jolie is considered the first language for microservices because it
    provides built-in primitives to handle service programming concerns without
    needing external frameworks or libraries.
    ...
```

We used Jolie's documentation[3] to create QnAs for every part of the language's documentation.
ILAB provides a tool to validate the different QnAs using the following command:

```
ilab taxonomy diff
```

Once we created all the QnAs for the documentation, we used ILAB to convert the taxonomy into datasets.

3.3 Dataset Generation

This is where the SDG parts come into play. ILAB uses an instructor LLM to generate datasets by expanding the taxonomy into larger text files, as discussed in section [2.2.2]. In our case, we used *Mixtral-8x7B* as the teacher model — the default teacher model for ILAB — which can be downloaded using the following command:

```
ilab model download --hf-token=*
```

Generating datasets on our Apple machine was, to say the least, *buggy*. We chose to use the HPC to run this task, which took almost 60 hours on an RTX 2080. If the tool had worked properly, we could have run it on our own machine as well.
To generate the datasets, we used the following command:

```
ilab data generate --pipeline simple
```

The "simple" flag for the pipeline is used for training on Apple machines.

We can now move on to the training phase.

[3]https://docs.jolie-lang.org/v1.13.x-git/introduction/index.html

3.4 Training the Base Model

Once the datasets were ready, we used the following command to start training the base model:

```
ilab model train --pipeline simple --local --num-epochs * --iters *
                          --optimize-memory
```

We ran this command multiple times to create several models with different iteration counts. We chose not to change the number of epochs — always set to 1 — because the process was somewhat buggy. For our case study, we settled on 100, 250, 500, and 1000 iterations.

Once all the models were created and ready to use, the final step was to convert them to *.gguf* format to quantize them.

3.5 Model Quantization

This step was necessary in order to *serve* the models, as there is currently no way to run safetensors files on Apple machines. The only available runner is **LLama.cpp**, which requires a .gguf file [2.4]. For each conversion, we used the following command:

```
ilab model convert --model-name jolieLLM
```

The last, and most interesting, step was to chat with the newly created models.

3.6 Serving and Chatting with the Models

ILAB allows you to chat with the model by serving it. You can do so using the following command:

```
ilab model serve --model-path ./jolieLLM.gguf
```

Once the model is being served, you can chat with it using the following command:

```
ilab model chat
```

4 Evaluation of the models

Once we got all the different models ready, we moved on to compare them to see how they perform as the iteration count increases. We opted for a straightforward method, which can be summarized with:

- We prepared five different text prompts, with different domain coverage (theorical, technical, hybrid and coding aspects of the language)
- For every model – even the base one – we asked the same prompt three times and saved their responses
- All of us gave a grade from 0 to 5 to every response

After this process, we took all the data and merged them into different plots, to show the behaviour of the models and their perfomance.

4.1 Prompts

Prompt 1
How does Jolie handle REST API's? Can you explain how does the language work when implementing them?

Prompt 2
What's the difference between processes and sessions in Jolie? Can you expand a bit more on the sessions?

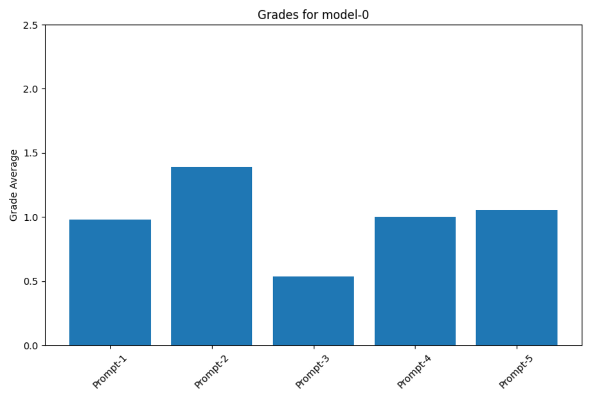
Prompt 3
How does Jolie handle database connections? Can you explain to me how that works and give me some practical examples on how to use them?

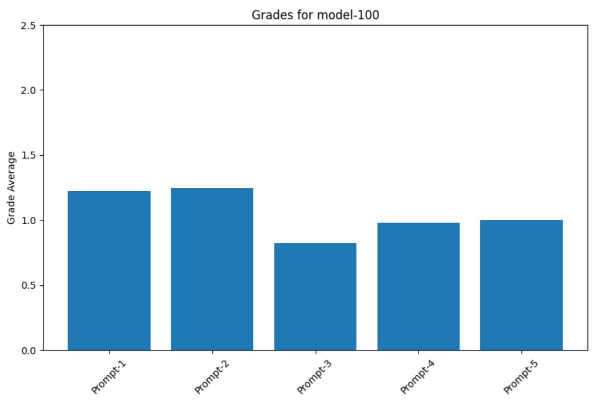
Prompt 4
Can you give me the code for two Jolie programs that communicate with eachother. I want to send an "Hello World" message from one to the other.

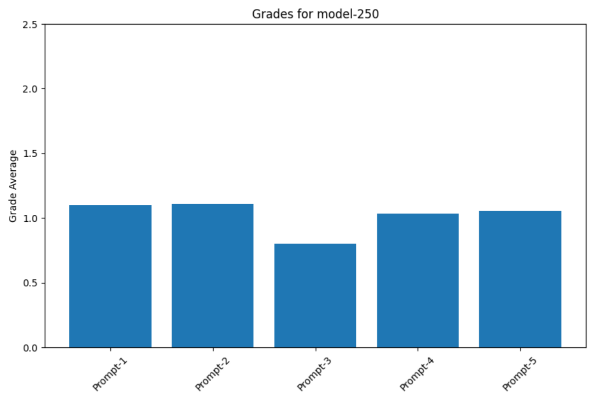
Prompt 5
Help me write the code for a calculator in Jolie. I want my calculator to be able to do basic functions, like sum, subtraction, divide, multiply. Give me the code for the Calculator and the code to try and communicate with it.

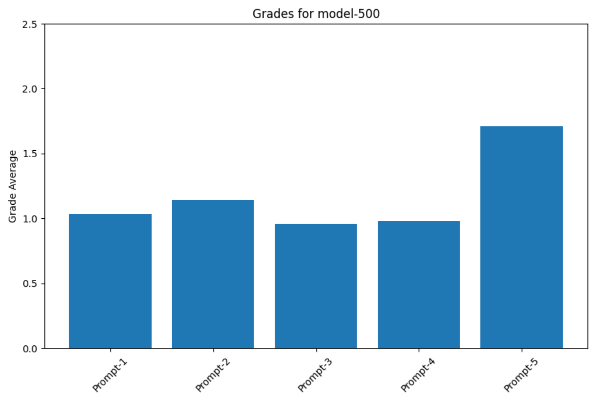
4.2 Results

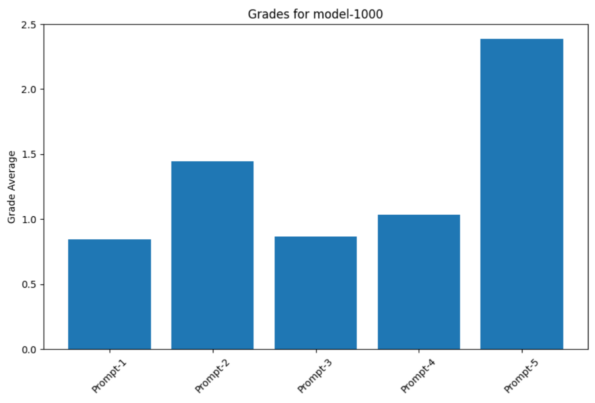
The process will be the same for every plot. We took every grade we gave for the prompt and made an average.

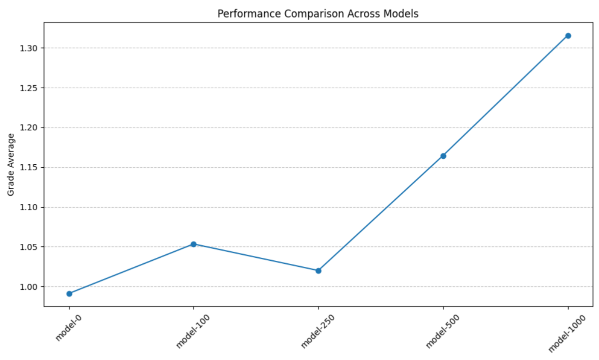


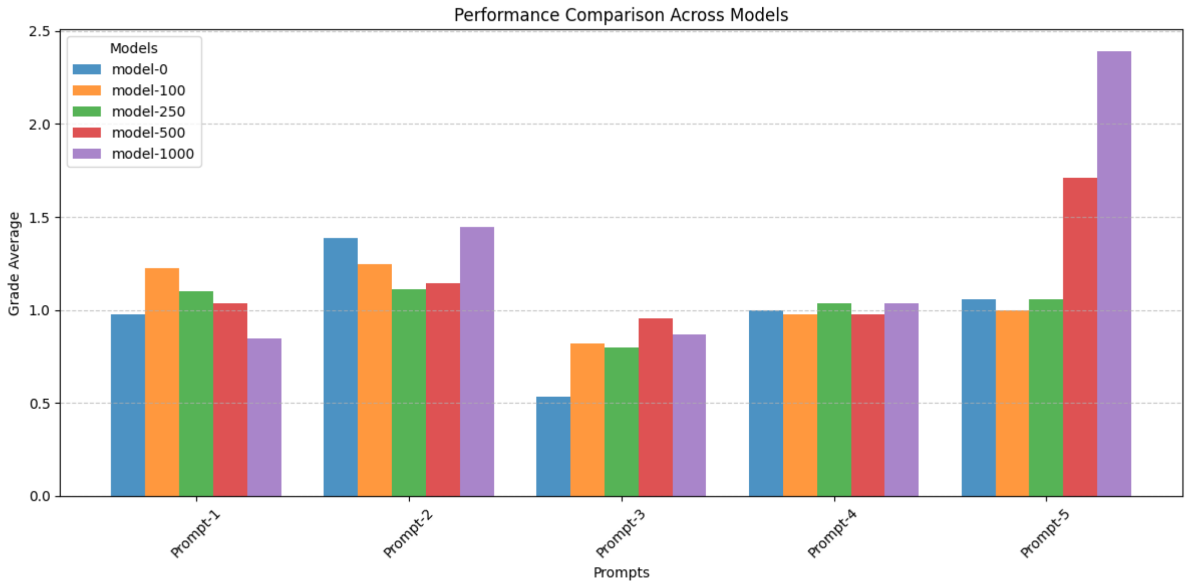












The first five plots represent the performance of the models based on the given prompt. We expected a linear growth, but we can clearly see a drop of precision with the 250 iteration model.
We can observe an overall improvement by looking at the mean performance plot, where the linear progression can be seen.
The last plot shows a comparison between all the prompts across the different models.

5 Bonus: a new model with 15K iterations

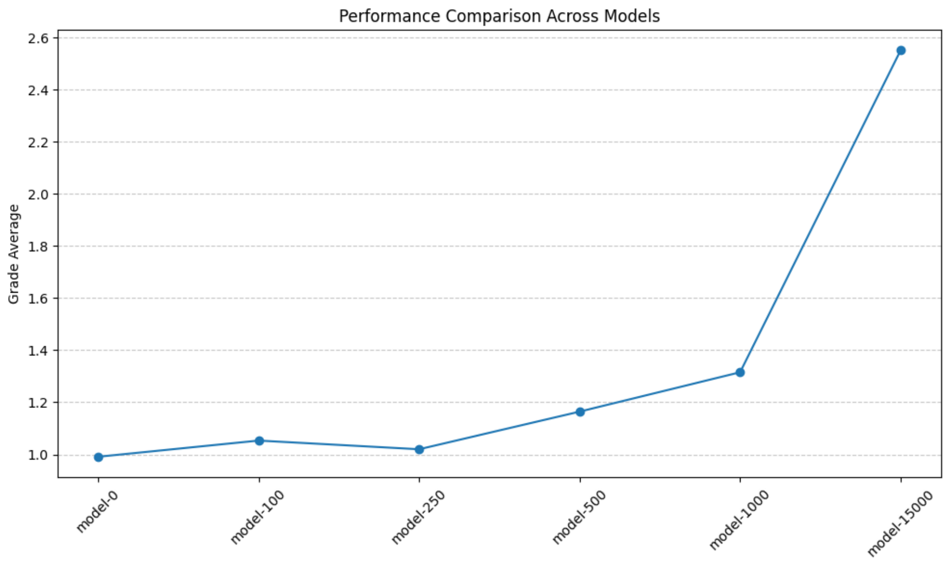
In the ILAB documentation, we encountered a particularly valuable resource: a Colab Notebook designed to facilitate the demonstration of the model fine-tuning process. This notebook provided a clear, step-by-step illustration of the underlying operations that occur when the command:

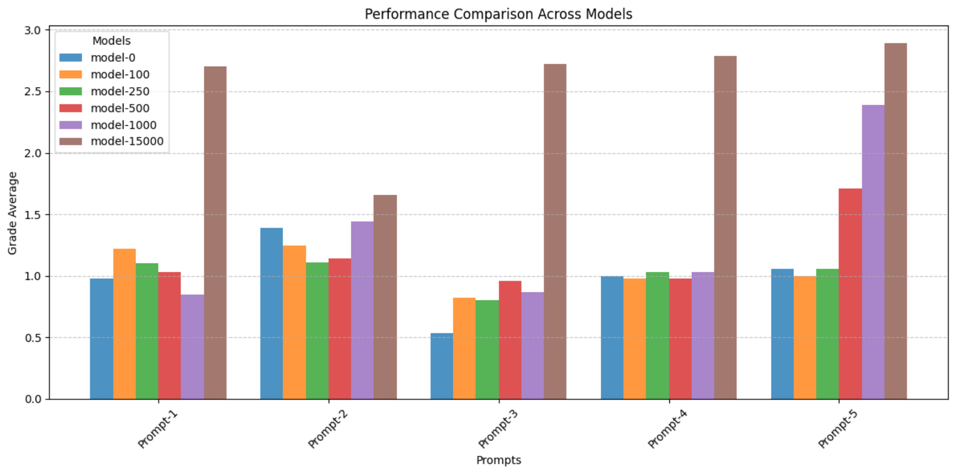
```
ilab model train
```

is executed. The practical utility of this resource cannot be overstated, as it allowed us to gain a deeper understanding of the internal workings of the tool, particularly during the training phase. Observing the process in action enabled us to better comprehend the intricacies of model refinement, including the impact of different hyperparameters and the iterative nature of the training process.

Moreover, leveraging this notebook proved instrumental in enhancing the quality and precision of our model. By executing the fine-tuning process on a more powerful GPU (an Nvidia T4 provided by Google), we were able to accelerate the training significantly. This not only improved the efficiency of the model development cycle but also empowered us to perform more extensive training, ultimately achieving a model with 15,000 iterations.

The 15k iteration cap was chosen because the training loss after this number was starting to grow again.
We did the same grading as before for this new model, and got these results:





It is pretty clear that with more iterations - and with more computing power - you can achieve better results.
Of course this goes beyond our initial goal, but we wanted to see how the tool would perform with more power.

6 Trying to visualize the attention mechanism

In the context of transformers and attention mechanisms, attention refers to how the model decides which parts of the input sequence to focus on when making predictions. It's a key feature of models like GPT-2, BERT, and others, enabling them to understand relationships between words (or tokens) in a sentence, regardless of their position.
In more technical terms, attention allows the model to weigh the importance of different words in a sequence relative to each other.
For example, in the sentence "The cat sat on the mat," when predicting the word "sat," the model may focus on "cat" because they are strongly related, while ignoring less relevant words like "on" or "the."

It could have been interesting to measure the attention maps of our models while performing inference, in order to see if the model is allucinating or understands the concept of what he is asked.
Using the `shap` library we were able to measure the attention on a standard not quantized LLM (like GPT2). When we tried to do the same with our models, we encountered some issues.

Measuring (or "reading out") attention weights in a transformer is normally done by having the model output its intermediate activations. In full-precision models, this is straightforward, but many quantization methods - especially those implemented for inference efficiency like bitsandbytes' 4-bit (like ours) or 8-bit quantization - fuse operations and replace standard linear layers with custom kernels that do not necessarily expose the intermediate attention matrices.

In other words, while the overall architecture (and even its outputs) remains the same, the very purpose of quantization is to optimize the forward pass (both speed - and memory-wise) by "compressing" the weights and often fusing computations. This usually means that the intermediate values (such as the un-normalized attention scores) aren't available in an easy-to-extract form.
While it isn't theoretically "impossible" to measure attention on a quantized model, the way current quantization libraries (such as bitsandbytes) are implemented means that the necessary intermediate activations (the attention weights) are typically not available. To make it work you'll likely need to either work with a full-precision model (not our case) or design a custom solution that "dequantizes" or bypasses the fused operations at the point where attention is computed (what we tried).

Assuming you can load a full-precision copy of your model - even if you originally used bitsandbytes to run inference with quantized weights - you can run a "full-model" forward pass and then apply SHAP on that version. In practice, this means **dequantizing a quantized model**.
Bitsandbytes also provides a method such as `model.dequantize()` that converts the quantized weights back to full precision. If you use that method, you obtain a dequantized model whose architecture mirrors that of the original model. In this full-precision copy, you can enable `output_attentions` and then compute SHAP explanations over the attention outputs.

Unfortunately, we are not able to use a machine with that much GPU VRAM, so while trying to make it work we encountered different `Out of Memory` errors, which blocked us from going forward.

With more powerful machines it could have been theorically possible to plot attentions maps!

7 Conclusions

ciao

References

[1] S. Sudalairaj, A. Bhandwaldar, A. Pareja, K. Xu, D. D. Cox, and A. Srivastava, “Lab: Large-scale alignment for chatbots,” 2024.

[2] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” 2021.

[3] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “Qlora: Efficient fine-tuning of quantized llms,” 2023.