Build Our Own Lima

Keya Hu
ACM Class 2021
hu_keya@sjtu.edu.cn

Yijin Guo
ACM Class 2021
guoyijin@sjtu.edu.cn

Hang Ruan
ACM Class 2021
zzrh01@sjtu.edu.cn

Abstract

This project utilizes methods from the Self-instruct paper to generate instructions, uses LLaMA-Factory to fine-tune the model Qwen1.5, and evaluates it via AlpacaEval.The evaluation results achieve a win rate of 23.16% in competition, indicating its practical value and potential for further development.

1 Introduction

In this study, we fine-tune and evaluate a language model based on Qwen1.5 through meticulously designed experiments. Initially, in the instruction dataset construction phase, we adopt the method proposed in the recent Self-instruct paper to ensure that the model can receive and understand specific task instructions. Subsequently, during the model's fine-tuning phase, we utilize the off-the-shelf tuning codebase LLaMA-Factory for supervised fine-tuning to align the model's expected behavior with its actual output, ensuring its accuracy on specific tasks. Finally, in the model performance evaluation phase, we use the AlpacaEval tool to conduct a comprehensive performance test of our fine-tuned model. We adjusted the parameters of supervised fine-tuning and conducted evaluations, compared the results, and conducted analyses. The better evaluation results show that it achieves a win rate of 23.16% in competition, indicating its practical value and potential for further development. Our repository is available here LLM alignment repository.

2 Methods

2.1 Automatic Instruction Data Generation

We follow similar steps in the paper Self-Instructed (Wang et al., 2022). We revise a few details to make it faster, more automatic and more adaptable for the GPT-3.5 version.

2.1.1 Instruction Generation

We separate the seed data instructions into two subsets: classification tasks and other generation tasks. We do this because if we sample them together, since there are many other tasks in the seed data, very few classification tasks will be generated, about 10 in 1000. Therefore, separating them into two subsets and randomly choosing to generate a classification task or not with the probability of their number in the dataset can produce a more balanced generated dataset.

In each subset, we use 6 tasks in the seed tasks and 2 tasks in the generated tasks, as an example in the prompt, to produce 8 new tasks in each API call.

We will filter out the following tasks:

1. The empty instructions.
2. The instructions contain images, graphs, and pictures that can not be solved by ChatGPT-3.5.
3. The instructions that are too similar to the existing tasks. We do it by filtering out tasks with a rouge score higher than 0.7 with any other existing tasks.

We generate 1000 valid instructions in total in this step since the paper says 1000 high quality instructions are enough for fine-tuning (Zhou et al., 2024). The code of this part is available here: generate_instructions.py.

2.1.2 Classification Task Identification

We randomly choose 12 tasks in seed tasks about classification and 12 tasks about others in seed tasks as an example of the LLM. We give these 24 tasks' instructions and whether its classification results are "Yes" or "No" in the prompt as an example, and tell it to generate whether the task is "Yes" or "No" a classification task of the newly generated task in last step.

We classify all the 1000 generated tasks and store them. There are 105 classification tasks among them. The code is available here: classify_instructions.py

2.1.3 Instance Generation

We use two different strategies to generate instances of classification tasks and other tasks.

1. Output-first approach

   We do this for classification tasks. We first generate potential class labels, followed by the conditional generation of respective input data based on each class label.

2. Input-first approach

   We do this for other kinds of tasks. First, the language model is required to determine the input fields based on the instructions and then generate the corresponding output.

We generate one or more responses to each instruction and put it into the final generated dataset. After generating these instances, we filter out the responses that are in the wrong format or are empty. Then we also add seed instructions in the generated instructions, accounting for total of 1134 instructions. The final generated instructions code available here generate_answer.py and filter_generated_task.py.

2.2 Supervised Fine-tuning with Generated Instructions

Dataset Our dataset is the generated data in the first step and we preprocess it to make them feasible for the tuning codebase. Since there are multiple inputs and outputs in each instruction, there are 2107 instructions in total after preprocessing.

Base model We utilize the recommended Qwen1.5 as our base model.

Tuning codebase We utilize the LLaMA-Factory as the tuning codebase. We set the batch size to 64(2 GPUs, each with a batch size of 32) and the number of maximum steps to 100. More detailed hyper-parameters can be found here .

It is worth mentioning that we have modified the maximum steps from 5000 to 100 (The hyper-parameter save_steps and warmup_steps are modified correspondingly). This will be introduced in detail in the Part 3.4.1.

Figure 1: Step=100

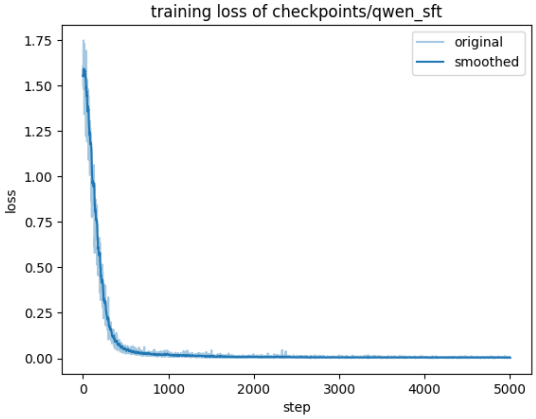

Figure 2: Step=5000

2.3 Evaluation

We utilize the alpaca-eval to evaluate our model. It uses chatGPT to make a preference of the answers generated by our fine-tuned Qwen-1.5 model and a reference powerful LLM(GPT-4).

The win rate means that for what percentage of instructions, chatGPT prefers the answers of our model to the reference model.

3 Results

All the results can be found in our GitHub repository.

3.1 Instruction Generation

We construct a generated dataset of 1134 instructions, which then be separated into 1134 instruction-input-output pairs. There are totally 137 classification tasks among 1134 tasks. The final generated instructions are available here.

3.2 Supervised Fine-tuning

Considering we have attempted two hyper parameters, we save both of the checkpoints.

The training loss in our fine-tuning process is shown in Figure 1 and Figure 2.

The two results' checkpoints are both available here, including many records and results of our fine-tuning. It's worth mentioning that the file "model.safetensors" is saved in SJTU Cloud Disk and can be found in our README.md.

3.3 Evaluation

Table 1 shows the different results of our evaluation with different max_steps. It achieves a win rate of 23.16% in competition when max_step=100 and 14.68% when max_step=5000.

More results of our evaluation process are available here.

Table 1: Evaluation Results

max_steps	win rate	average_length
5000	14.68	7167
100	23.16	2428

3.4 Further Experiment

We analyze our results and do some more experiments. Here are our findings.

3.4.1 Fine tuning

In the fine tuning process, we initially set the maximum steps (max_steps) to 5000, which is given in the reference repo. This setting finally achieves a win rate of 14.68%. Then we modify the step to 500(the warmup_step is modified to 10 correspondingly) and improve the result to 23.16%.

Here is our analysis for the phenomenon. The original setup corresponds to approximately 152 epochs (with total data of 2107 instructions and a batch size of 64). The results show that the adjusted parameter settings significantly improve the evaluation outcomes of the model. This suggests that fewer training epochs might help remit overfitting and allow the model to converge to an optimal state more quickly in this specific task and dataset. Additionally, the minimal warm-up steps reduce early training learning rate fluctuations, which might also contribute to enhancing model performance. Therefore, we recommend flexibly adjusting training parameters based on the characteristics of the data to achieve optimal results in similar tasks.

3.4.2 Evaluation

The evaluation process can be divided into two parts. The first part is to output the answers of our model and the reference model given some same prompts. The second part is to grade the two outputs using chatGPT and compare the "grades".

After the first part, we can obtain the outputs of our model (the file model_outputs.json). We find that it somehow demonstrated a weird behavior of asking and answering itself. That is to say, the output will be "assistant: {correct answer} system:{you are a good agent} user: {a new question} assistant: {answer to new question} ...". Therefore, we manually extract the correct answer to the instructions and discard the question-asking and question-answering part, which makes the output of the instructions more reasonable. The processed output is available here: model_outputs_processed.json.

Now we do the evaluation again on the new processed data. Here, we share the fine-tuning results of max_step=5000, which achieves a win rate of 14.68% initially. After rerunning, however, the win rate becomes 9.69 %, which is lower than before. We guess it's because the length of the answer really matters since the average length of results decreases from 7167 to 2428. The unprocessed output gains a discrete win rate of 14.68% and a length-controlled win rate of 11.74%. The processed output gains a discrete win rate of 9.69% and a length-controlled win rate of 9.37%. Therefore, we can see that with the process of extraction, the true correct answer can reduce a decline in the length-controlled win rate.

Note that, with 100 maximum steps fine-tuning, this asking and answering it self phenomenon has greatly reduced, which we guess account for part of why its win rate increase.

4 Division of labor

Keya Hu:

- Responsible for the Instruction Generation (including the relevant part in the mini-paper).
- Operate the second experiment(Part 3.4.2).
- Maintain our GitHub repository.
- Help with the api key in evaluation.

Yijin Guo

- Responsible for the Fine-tuning part. (including the relevant part in the mini-paper).
- Run the pipeline of fine-tuning and evaluation and normalize the process into script files(Shell).

Hang Ruan

- Responsible for the Evaluation part (including the relevant part in the mini-paper).
- Run the pipeline of fine-tuning and evaluation, adjust parameters, and optimize results.

Acknowledgments

Thanks to TA Fan Zhou, Shijie Xia, and Yixiu Liu for their help, and thanks to our teacher Pengfei Liu for teaching. Thanks to the financial support of api key from our teacher.

References

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2024. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36.